TAMPERE UNIVERSITY OF TECHNOLOGY

TERO MAARANEN

# Implementing a Multithreading Framework in C++

Master of Science Thesis

Examiners:
Prof. Tarja Systä (TUT)
Prof. Kai Koskimies (TUT)
Subject approved in department
council meeting on 8.6.2011

# TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
**Tero Maaranen: Monisäikeistävän C++-ohjelmistokehyksen toteuttaminen**
Diplomityö, 49 sivua, 6 liitesivua
Heinäkuu 2011
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Tarja Systä, Prof. Kai Koskimies
Avainsanat: Säie, C++, ohjelmistokehys, suunnittelu, toteutus

Monisäikeistä ohjelmistoa suunniteltaessa ja toteutettaessa on usein järkevää aloittaa luomalla ohjelmistokehys, joka hallinnoi toistuvia, hankalasti hahmotettavia ja monimutkaisia rinnakkaisuuteen liittyviä toimenpiteitä. Kehyksen luonti saattaa itse asiassa olla ohjelmistoprojektin aikaa vievin osuus, ja itse toiminnallisuuden lisäys onkin vain viimeinen silaus muuten valmiiseen järjestelmään.

Tässä diplomityössä esitellään säikeidenväliseen kommunikointiin tarkoitetun ohjelmistokehyksen luontiprosessia ja sen toteutukseen vaikuttaneita syitä. Tavoitteena oli toteuttaa pienessä mittakaavassa toimiva mutta mahdollisimman monta käyttötarkoitusta tukeva ohjelmistokehys, joka ei kuitenkaan olisi tarpeettoman tehoton. Lisäksi tarkoitus oli pyrkiä kehittämään monisäikeisten ohjelmien ymmärrettävyyttä ja vähentää niiden monimutkaisuutta. Ohjelman varsinainen käyttötarkoitus on mahdollistaa monisäikeisen pelimoottorin vakaa ydin.

Niinkin monipuolisessa aiheessa kuin monisäikeinen ohjelmointi ei ole olemassa yleispätevää ratkaisua, joka selvittää kaikki kehittämisongelmat. On kuitenkin mahdollista löytää tiettyjä toimintatapoja, joita käyttämällä voidaan merkittävästi helpottaa osaa ongelmista.

Työssä on keskitytty täysin yksiprosessiseen ohjelmistoon, mutta muokkaamalla ja uudelleenkäyttämällä olemassa olevia rajapintoja, kehyksen voi saada toimimaan useiden prosessien välisessä kommunikoinnissa.

Työssä on mitattu tehokkuutta perinteisestä monisäikeisestä ohjelmasta ja vastaavan toiminnallisuuden sisältävästä ohjelmistokehystä käyttävästä ohjelmasta. Tuloksista saa käsityksen, että kehystä käytettäessä tehokkuus on heikentynyt merkittävästi. Lisäksi erityisesti pienissä ohjelmissa ohjelmiston lähdekoodin luettavuus - hieman yllättäen - kärsii monimutkaisemmasta rakenteesta ja kehyksen käytön aiheuttamasta lisästä. Voidaan kuitenkin olettaa, että suuremmissa ohjelmistoprojekteissa, joissa perinteinen monisäikeinen ohjelmisto saattaa muodostua äärimmäisen monimutkaiseksi, ohjelmistokehys selkeyttää ohjelmaa tehokkuuden kustannuksella.

# ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Program in Information Technology
**Tero Maaranen: Implementing a Multithreading Framework in C++**
Master of Science Thesis, 49 pages, 6 appendix pages
July 2011
Major: Software Engineering
Examiner: Prof. Tarja Systä, Prof. Kai Koskimies
Keywords: Thread, C++, framework, design, implementation

When designing and implementing multithreaded software, it is often justified to start by building a robust framework that handles the repetetive and complicated procedures required of a multithreaded application. This can be the most time consuming process, and adding the actual functionality on top of the framework can be considered only as the final touch.

In the thesis work, a multithreading framework has been designed and implemented from scratch. The goal was to build a relatively small framework which could be used for practically any multithreaded program. Yet, it was attempted to keep in mind some of the most important aspects of multithreaded programming, such as understandability, complexity and performance.

For such a complex topic as multithreaded programming, there is no "silver bullet" approach to solve all problems, but one approach can still fit to many needs. The implemented framework concentrates on single-process applications, but by reimplementing some parts of the framework, also inter-process communication can be made possible. The actual intended use for the framework is to form the core for a multithreaded game engine.

Performance analysis is provided in small scale by implementing well-known algorithms. The analysis shows that performance is clearly degraded compared to conventional multithreading solutions, and that, especially in small projects, code readability suffers from the more complex nature of framework usage. This implies that frameworks suits larger projects better, as in such projects the conventional approach tends to grow into an unmaintainable and messy source code.

# PREFACE

This thesis was written for personal interest in multithreading and implementation of generic frameworks in general.

Programming part of the thesis work began approximately in June 2009. The idea was to implement a multithreading framework that could be used in small game projects. Rough draft of the basic functionality was implemented quickly, but debugging and finishing had to be postponed due to long days at my day job. Then, finally, in June 2010 I contacted professor Haikala with this idea for a thesis. Following that discussion the final phase of programming continued rapidly. By the end of August 2010 the programming work was completed.

The oversight and analysis parts were mostly written from July to September of 2010. At this point I also started working on my candidate's thesis, which was written during the fall of 2010. Also, at the same time new responsibilities at Nokia virtually halted progress of the master's thesis again. In the end of March 2011 the thesis examiner was changed to Tarja Systä. This change in examiner also pushed me to get the correct writing spirit and encouraged me to finally finalize the thesis.

All in all, writing this thesis has been a long and stressful process. I'd like to thank professors Tarja Systä and Kai koskimies for providing insightful feedback and support concerning the work. But especially I want to thank my fiancée Iina for the never-ending moral support she provided during the years of coding, debugging and writing.

Tampere
July 6, 2011

Tero Maaranen

# CONTENTS

# LIST OF FIGURES

# 1. INTRODUCTION

People who design computers have always strived to make them faster and more efficient. The greatest breakthrough of computer industry was the invention of transistor, which practically exploded the efficiency of integrated circuits. Descriptive to this incredible advancement is Moore's law, which states that the number of transistors on an integrated circuit board doubles roughly every two years. This has been amazingly true so far, providing the possibility for ever faster computers.

However, the laws of physics are causing Moore's law to be insufficient in continuously providing the growth in speed of computers. It's getting more and more difficult to keep the clock rates and heat dissipation of the integrated circuits increasing.

In order to continue improving the computers' efficiency, it has been a trend of computing industry for some time now, that the new hardware and programming lean towards parallel computing. By adding parallelism into the computers, the hardware can simply add another processing unit next to the existing ones and gain more computing power.

Unfortunately, adding more processors does not make all the existing software faster automatically. Programmers need to explicitly start taking the existing hardware into use. Single threaded software can only use one processor at a time, so optimally, the software could be designed so that there is always a running thread for each processor.

It is a tremendous possibility for the programmers to make their programs more efficient, if only they can design programs that have orthogonally independent functionality that does not require constant synchronization with other threads. However, with the current support in programming language, adding new threads is also greatly adding complexity into the source code. What happens with the current programming paradigms is that when the computers have more than four processing units that should be taken into use, the programmer cannot effectively maintain the code anymore. Furthermore, the CPU manufacturers have already been announcing new processor architectures that consist of up to 64 cores [5]. In order to get full advantage of the hardware, it is required to study and implement more advanced techniques to take more processing units into use with as little complexity as possible.

For this thesis, a version of a somewhat multipurpose multithreading framework was implemented in an attempt to create a sophisticated and moderately efficient basis that can be used for large software projects, such as games or device drivers. Firstly, developers shall be able to create new threads in C++ simply by defining a class which inherits a threading class. Example of this kind of approach can be seen in Java, which provides a Thread class [8]. Secondly, communicating between threads shall be easy to implement, easy to understand and error-proof. It is expected that if developers do not need to explicitly protect data from corruptions caused by inter-thread communication, this objective can be achieved.

With this kind of framework the developer can implement multithreaded applications using C++ so they can easily divide different interacting logical components, yet making it possible to implement the source code almost as if it was single-threaded. It is the framework's responsibility is to take care of all logic of inter-thread communication as well as protecting data from corruption.

The thesis starts by explaining the basics of multithreaded programming in Chapter 2. Chapter 3 continues by revealing some of the most concerning problems that occur when designing and implementing multithreaded software. These concerns can be considered as the primary motivator for the thesis work.

The design guidelines, basic structure and future improvements of the framework are described in Chapter 4. Chapter 5 describes the APIs of the framework and explains what main principles should be followed when using them.

In Chapter 6, the framework is used to create a small program, which demonstrates a well-known multithreaded problem. Performance of the program is analyzed and compared to another implementation which is intended to have as efficient inter-thread communication as possible.

Finally, Chapter 7 concludes and summarises the problems and feasibility of the implemented framework for personal use in multithreaded applications and games.

# 2. MULTITHREADED PROGRAMMING

## 2.1 Processes

Quite simply, a process can be thought of as one program running in the computer. All modern operating systems are able to run several processes in parallel or simultaneously.

The operating system moves processes within different states. In short, the usual state diagram of a process consists of the following states: *created, running*, *waiting, blocked* and *terminated*. Whenever a new process is created it will be put into created state, from which it will be moved into waiting state, as shown in Figure 2.1. This means that during creation, the operating system will reserve resources for the newly created process. When the operating system's scheduler moves the process from waiting to running state, it allows the process to reserve a slot of processor time, thus allowing the process to execute. The process will be put back into waiting state when its slice of time is spent, and it will remain there until the scheduler again sets it running.

From running state the program can also enter either *blocked* or *terminated* states. When in blocked state, the process will remain sleeping until it is explicitly woken up. A sleeping process does not consume any CPU time, so it is the preferred way of making processes wait for external activity, such as key presses on keyboard. Entering the blocked state is usually done by *locking* a *semaphore* or a *mutex*. When the blocking primitive is *signalled* the sleeping process will be woken up by setting it to waiting state.

When process exits, it will be put into terminated state from which it will never recover. After process has been terminated, the operating system performs cleanup operations for it [7]. Creating and killing processes are heavy operations, so they should be limited to minimum. Rather than killing an old process and creating a new one, programs could make the processes wait in sleep state for signals that will wake them up.

There are also other states for processes, such as swapping out the process into mass memory. Those states are not in the scope of this thesis as they are not relevant from the thesis work's point of view.

An especially important aspect of processes is that they all have their own memory-space. From the process point of view, no pointer from another process –

regardless of its value – can point to the other process' memory location. No process has visibility to other processes' memory space. This means that inter-process communication requires support from the operating system. Operating system's kernel has visibility to all processes' memory spaces. Thus the operating system can relay the data between processes, as shown in Figure 2.2. The data being passed from one process to another has to be copied to the target process' memory. This may cause performance issues in the form of excessive copying of data, as well as out-of-memory issues as effectively same data is being stored in several memory locations.

Some simple operating systems – such as Microsoft DOS – have a single-process model, in which the user is allowed to run only one process at a time. Naturally, this is a severe handicap for implementation of multitasking applications, as any concurrency needs to be implemented separately into the program.

In the early days of multitasking operating systems, simple cooperative multitasking schedulers were implemented. Cooperative multitasking can be dangerous as it relies on the application to relieve control back to the operating system. Danger means that if the application misbehaves, by accident or on purpose, it can cause the whole system to a halt, making it extremely easy to implement malicious software.

Most modern operating systems implement pre-emptive multitasking scheduler, which allows the operating system to perform time-slicing which will automatically set the process to sleep after its slice is consumed. Also, with pre-emptive multitasking the operating system has the possibility of forcefully killing misbehaving applications.

It should be noted that time-slicing does impose some performance degradation, as switching between processes requires rewriting some or all of the processors registers. Therefore, the shorter the time-slices are, the more overhead there is. Generally, process context switch is considered as a heavy operation for the operating system.

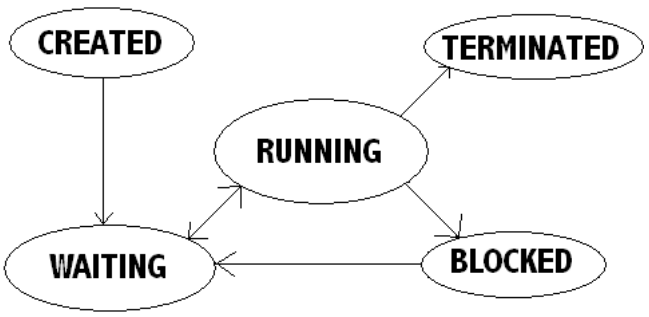


Figure 2.1: Process states

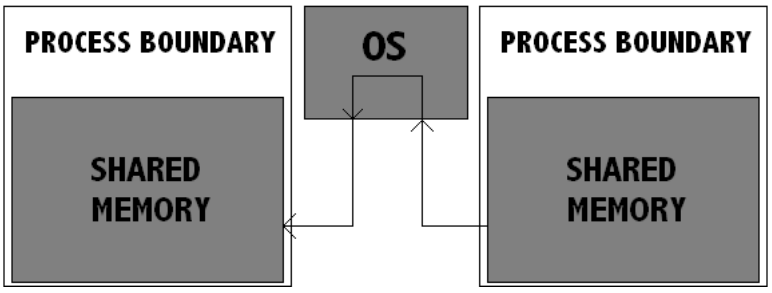


Figure 2.2: Inter-process communication

## 2.2 Threads

Depending on the operating system there can be multiple *thread*s running simultaneously in the process. A thread is often explained as a light-weight process, as context switching between threads is rather swift. If threading is not supported and yet programs are wanted to perform multitasking it needs to be achieved through multiple processes and inter-process communication.

If the operating system does support threading, it makes it possible for an application to run several threads, or code branches, at the same time. For a single-processor computer the simultaneousness is an illusion as the threads are actually run in series, but to the end-user it seems parallel because the threads get swapped so quickly. The swapping happens in the same manner as time-slicing described in Section 2.1. Then again, in a true multicore-processor threads are really run simultaneously – to a certain degree due to synchronization needs. Synchronization will be covered later in Chapter 3.

In different operating systems and different programming languages threads are handled in different ways. In this thesis threads are considered as they are defined in Posix thread specification and as they are implemented and used with *Pthreads*-library for C++ [2]. Pthreads is a platform independent API that provides a standardized C-interface to the most critical multithreading operations.

In POSIX, threads are considered as "an independent stream of instructions that can be scheduled to run as such by the operating system" [2]. They are run inside the same process and they have the same memory space. Each thread has a unique stack pointer, registers, scheduling properties, signals and thread specific data. But they all exist inside a single process and its memory-space.

Process context switch requires reloading the PCB [7]. Thread context switching is a much lighter operation as it happens within the process. Process only needs to switch stacks to switch to another thread, not all the other registers.

Threads of a single process can access the data of other threads due to the shared memory-space [2][17]. This allows for efficient inter-thread communications, although it brings synchronization responsibilities to the programmer. [12] See Figure

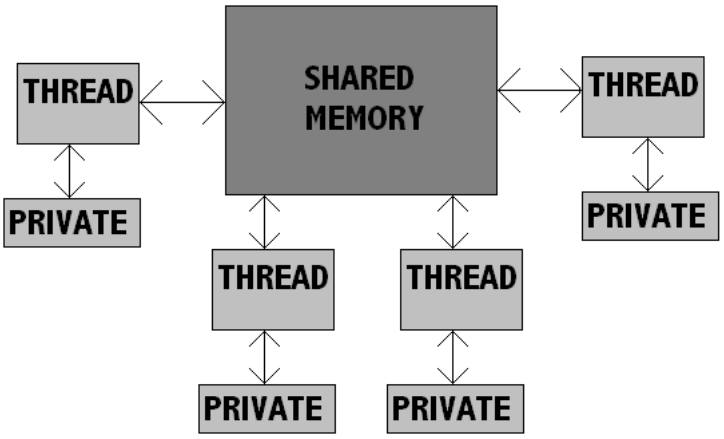


Figure 2.3: Process with threads

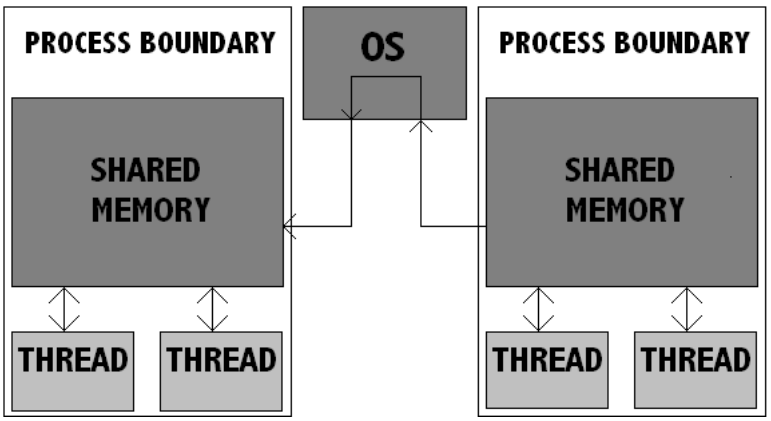


Figure 2.4: Threads and processes

2.3 for a picture of a process that has shared memory and multiple threads that may access it, and Figure 2.4 for two processes that share memory via operating system calls. In the latter case the processes cannot directly access each others' memory.

## 2.3 Synchronization

Whenever threads have shared memory that can be accessed at the same time, the data should be protected. Accessing unprotected data may cause data corruption and potentially crash the program. Updating memory may not happen in an atomic operation, so a memory reading thread may actually read a value that is not fully updated [7]. This topic will be addressed more thoroughly in Section 3.2.

There are different techniques to have threads synchronized. In this thesis mostly mutexes and semaphores are focused on. Mutexes and semaphores act in a similar manner in different programming languages. According to Symbian Developer Library the behavior of a semaphore is defined as follow: *"A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number.*

*Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore)."* [16]

In the same documentation mutexes are compared to semaphores *with the following description: "Mutexes are typically used to serialise access to a section of re-entrant code that cannot be executed concurrently by more than one thread. A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section." An easy way to grasp the concept of a mutex is to think of it as a semaphore with a starting value of 1.* [16]

# 3. PROS AND CONS OF THREADING

## 3.1 Advantages

Today, most powerful CPUs have dual- or quad-cores. It has been predicted that with the current trend of adding more cores to the processors, in the near-future it will not be uncommon for computers to have 32, 64 or even 128 cores. If that is the case then a serious effort is needed to be able to make use of those cores. In order to help programmers, the latest version of the C++ standard finally defines threads as a part of the language[15].

Using a C++0x compliant compiler the programmer does not have to use external libraries to achieve multithreading. This is potentially an incredibly important step in providing programmers more solid ground on which they can continue creating robust multithreaded applications, as they do not have to rely on threading libraries. Allegations have been made that threading libraries cannot be thread safe, ever [1][3]. Still the added complexity of multithreading is going to force applications and libraries to be relatively single-threaded even with the latest additions.

If an application only takes use of a thread or two, there could be dozens of cores idle as a couple of cores are overloaded with heavy operations. The single thread will perform large operations causing memory bandwith to be bogged down when large amounts of data are transferred from memory to CPU. Then again, there can be relatively long times during which memory is not accessed at all as there are so few threads preparing data to be sent. If the application could divide its work evenly for all cores, it could distribute the bandwith needs to smaller and more evenly distributed workloads. With this kind of approach the application, and the whole system, could speed up significantly [11].

In the past most obstacles with speed has been dealt simply by increasing the frequency of the CPU. This causes a slow application with design bottlenecks to simply get faster, even though the root-cause of the slowness is not fixed. It is expected that this kind of performance improvements cannot continue for long as frequency cannot be increased much further [13].

## 3.2 Difficulties

So far, there have been mostly positive aspects considered when speaking of multithreading, although there are also loads of problems related to using them.

Primarily, when considering adding threads to programs it should be noted that applications tend to become an order of magnitude more complicated per new thread [4]. By adding new threads the programmer will quickly change the source code into an unmaintainable software. Also, often there will be errors that are especially difficult to find and that cause random crashing of applications or simply hangs the system.

The fact that threads are capable of accessing each others' data within the process causes a great risk. If the accessed data is not protected by a synchronization primitive, such as a mutex, two threads may end up reading and writing into the same memory location simultaneously. This can, and usually will, lead to data corruption. This is referred to as *race condition* and it can be fixed by carefully placing the parts of code that are endangered within protection. It can be a very tedious process, as sometimes larger parts of code need to be protected as a whole, also keeping in mind that this kind of source code cannot be re-entrant or cyclic. Otherwise there may be a *deadlock*.

A deadlock is a situation in which there are two, or more, threads waiting for each others to release a protection primitive to be released before continuing. Neither thread can release the primitives they hold until the other one releases first, and therefore the system is in a deadlock. The situation is often compared to two trains that are waiting for the other one to pass a crossing before being able to continue, see Figure 3.1 for a visual illustration of the issue. Trains A and B can be regarded as threads that are waiting for a mutex to be unlocked. Avoiding this situation requires careful planning and design of a multithreaded program.

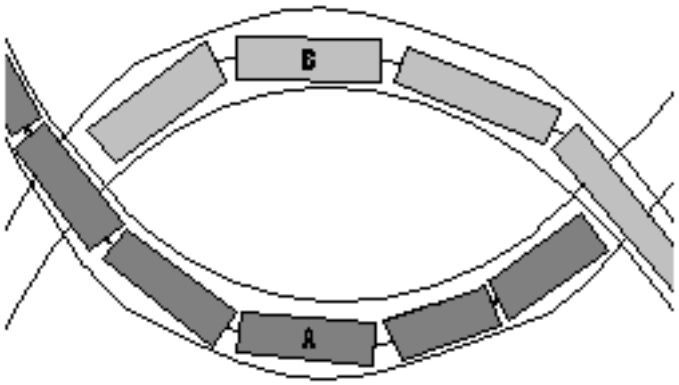


Figure 3.1: Deadlock analogy

Priority-inversion is a different kind of error that can cause a high priority thread to be kept in waiting state while lower-priority threads are being given processor-time to execute. It is caused by a high-priority thread locking a synchronization primitive, which should be unlocked by another thread. If the other thread is a low-priority thread it might never get a chance to run and release the lock. In Figure 3.2 thread A, which is a high-priority thread, is locking a mutex, which should be unlocked by the low-priority thread D as soon as D gets to run. In this scenario, there are also middle-priority threads B and C that keep getting into run state instead of D. Only after B and C are being put out of run state for some reason, for example, waiting for a keypress, thread D gets chance to release the lock. Only now the highest priority thread gets to run again. This kind of scenario can be extremely hard to detect simply by reading the source code as threads may be created all over the code.

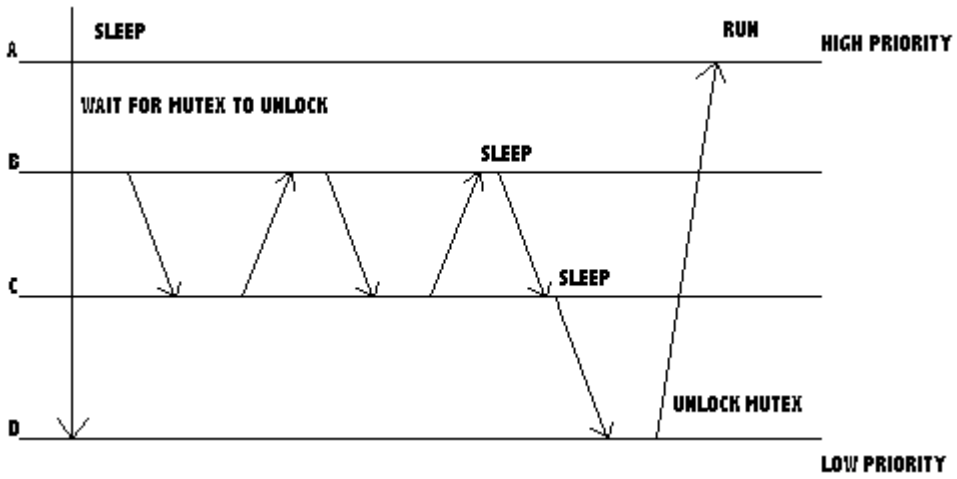


Figure 3.2: Priority inversion

If inter-thread communication is considered as a simple function call that will return a value of some type, there are again few issues to deal with. First of all, it is possible to call the other thread so that the actual call creates an object of the return value type. The implementation of such a function, say `interThreadCall()`, has to do a safe thread switching and save the return value from the other thread, and then again safely return it. It is a difficult thing to implement robustly, and definitely difficult to maintain if changes are needed. If the language or library used has a thread message functionality, it makes things a lot easier, but if not, it would require an implementation of a inter-thread message queue for sending and receiving the data. Implementing a message queue requires precise handling of mutexes or semaphores.

One of the easier ways of returning the return value is to use a *callback* function. A callback is either a dynamically bound method or a simple static function whose

address can be passed as a function pointer. Callback is usually used to issue control back from a service to the caller. [10]

In the context of the thesis, a serving thread calls the callback function and sets the other thread's reserved memory location of the return value to the new value. In multithreaded application the problem is that it requires protection of the memory, which needs to be done with a mutex or a semaphore.

Also, programmer has to be especially careful not to start executing the original caller's code in the serving thread. Such calls may easily end up into cyclic calls and deadlocks.

Basically any communication between threads requires a very high amount of protection and it will get extremely hard to determine the most efficient way to use the protection primitives, and extremely hard to thoroughly understand the written source code. All of this makes the written source code that much more difficult to maintain.

# 4. REQUIREMENTS FOR THE MULTITHREADING FRAMEWORK

## 4.1 Frameworks

A framework can be considered as a chassis into which users can add parts of new code. Often the framework by itself is not a functional program at all, but it becomes such when completed with the user code. The framework and the added parts together form a fully functional program.

According to their book, Koskimies and Mikkonen tell that the primary objective of frameworks is to help create reusable components. Reusing of components does not limit to source code as it includes also the architecture, which makes it easier to understand the components and add new functionality to them. [10]

Frameworks are generally divided into two main categories, white-box and black-box frameworks. White-box frameworks are generally considered as software components that provide complete visilibity to the inner functionality of the framework. Black-box is the opposite, so the user only has access to the APIs. Black-box frameworks require well documented APIs as developers cannot trace the reasons for unexpected behavior into the framework.

## 4.2 Requirements

In order to tackle the most important problems with multithreaded programming, the framework should be designed so that the user could implement software almost as if the program was single-threaded. In the design and implementation of the framework, the following points shall be considered.

1. Simplifation of source code
2. Minimization of mutex-handling
3. Protection of shared data
4. Request sending shall return immediately
5. Embedded requests
6. Differentiating of signals and callbacks

7. Simultaneous requests

The requirements are listed in priority order. Following sections will explain the reasons and advantages of the above-mentioned requirements in detail.

### 4.2.1 Simplification of source code

First of all, all the code shall become easier to read, understand and maintain. In this thesis complexity is regarded as the number one problem in multithreaded software, because it is so easy for complexity to explode so that larger programs become too difficult to maintain. The framework shall be implemented as a black-box so that developers do not need to understand the actual implementation but only the API.

Whenever programmers need to implement code with multiple threads, they always have to understand in which thread a certain code-block is executed. If an application has common code that is run in many threads, the maintenance and understandability of the code becomes more difficult. If the code is run in more than two threads, there's already a very high risk that there could be protection and synchronization errors.

The well-known "Keep It Simple Stupid"-idiom shall apply for the multithreading framework. Whenever programmer writes code for a thread that part of the code shall be run in exactly one thread. It is the framework's responsibility to restrain the execution of a thread from entering another thread's code.

In order to create a very simple shared data structure, programmer needs to create a new `ThreadQueue` object and define getter and setter interface objects and the programmer has a fully protected data storage. Creating a `ThreadQueue` shall be as easy as inheriting from `Thread` base class in Java [8]. The implemented data storage may be accessed by any number of other threads without any extra effort. The interface can be identical to a standard C++ class for the user, it has a 'getter' and a 'setter' that can be called. See Figure 4.1 for a graphical view of the storage.

However, the implementation of the storage requires a small addition: the call to the interface needs to be routed to the framework, see Figure 4.2 for understanding how to achieve this. The call to the framework is performed by calling `AddRequest` method of `ThreadQueue` base class. Simplest form of request routing can be

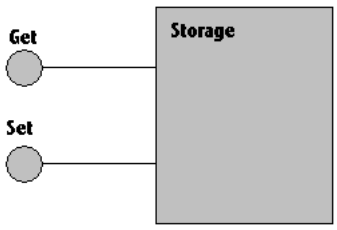


Figure 4.1: Interface of a thread-safe storage

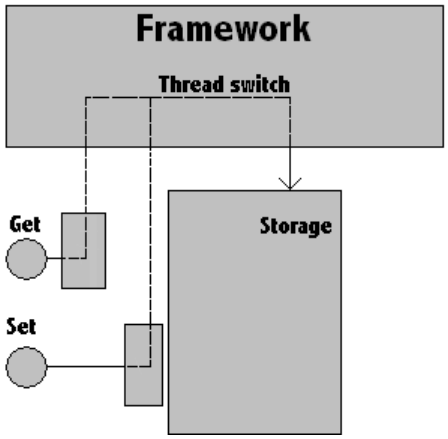


Figure 4.2: Storage interface explained

accomplished with one line of C++ code, so it is rather simple to understand. See Figure 4.3 for an example code for how to implement such request sending. `Producer` inherits `ThreadQueue` and holds pointer `iStorage` to a `Storage` object. `Storage` class also inherits `ThreadQueue`. `ThreadQueue` implements method `ThreadQueue::Add()` which can be used to issue request for that thread to serve. The implementations of `Storage::Get` and `Storage::Set` methods create new requests, cast them to `BaseRequest` pointers and pass the pointers to `ThreadQueue`.

The idea behind the request approach is that when adding new functionality, a new type of request is defined instead of adding a new method into the API. For example, if the `Storage` class is supposed to be extended so that it provides an API for returning the count of stored items, traditionally a new public method is added. This will cause binary incompatibility between different versions of code, which is a serious problem [14]. It is a problem especially if either the user code or the implementation of the API is not available, in which case the API simply cannot be changed. The requirement for the framework states that the public API of `Storage` class is not changed, but instead a new request type is defined. When the new request is inherited from `BaseRequest` it can be passed via the framework to `Storage`. This way developers can add new requests without ever breaking the binary compatibility of components.

Creation of new requests is explained in detail in Chapter 5. What is noteworthy right now is that the `Producer::Handler()` function is easily understandable, and everything acts as if it's written as single-threaded code. When `iStorage->Get()` returns, a request has been appended into Storage's thread and execution immeadiately returns. When Storage completes serving the sent request, a callback will occur so that execution automatically enters `Producer::Handler()`, again in the correct thread. User code can use an integer member variable, iPhase is used

in this case, that will be incremented each time the handler is called. This phase number can then be used to distinct different states of the asynchronous component.

Another solution for creating requests would be to skip this extra abstraction layer and cast the `ProduceRequest` pointer to `BaseRequest` in Producer's code and call `ThreadQueue::Add(arg)` directly. The problem is that it would break the modularisation idiom and it would cause code duplication, so the first example shall be considered as the primary coding convention.

```
void Producer::Handler()
{
  iPhase++;
  switch( iPhase )
  {
  case 1:
    // call the framework wrapper
    iStorage->Get( this, 0 );
    break;
  case 2:
    // read the return value
    // with ThreadQueue::ReadCallback()

    // reset iPhase
    iPhase = 0;
    break;
  }
}
// framework wrapper created the request and
void Storage::Get(ThreadQueue* aSender, int aIndex)
{
  GetRequest* req = new GetRequest( this, aSender );
  auto_ptr<BaseRequest> reqP( req );
  // make auto_ptr the only reference to the request
  req = NULL;
  // send to framework, returns immediately
  Add( reqP );
  return;
}
```

Figure 4.3: Sending an inter-thread request

### 4.2.2 Minimization of mutex-handling

One of the most difficult things for programmers is to understand all possible situations where a thread needs to be synchronized and which data should be protected, as it was shown in Section 3.2. If a framework could minimize or optimally diminish completely the need for mutexes, it could potentially be a huge advantage for the programmers.

In practice it would mean that the framework itself is implemented using mutexes and semaphores, but the programmer who is using the framework, would never have to worry about them. The framework will need to implement a thread-safe message-queuing algorithm for passing requests to each others as well as a thread-safe way for returning the return values from the other threads.

If synchronization problems would still be found, the problem would be inside the framework. The framework is likely a much smaller piece of code than the rest of the software being developed. Also, when the framework is maturized, most of the errors will be fixed from it, allowing much more robust code to be written. This way the programmers can create much larger applications but still have the advantage of keeping the code less prone to errors.

By sending all the requests through such a framework actually makes it possible to create even complex multithreaded applications without any mutexes maintained by user code.

### 4.2.3 Protection of shared data

There are two main reasons to use mutexes and semaphores in a thread. Most of the time the threads need some kind of synchronization between them, usually to protect shared data.

Let's take the classic producer-consumer problem as an example. In this scenario two or more threads share a fixed-size buffer so that producer threads are adding data into the buffer and the consumer threads read data from it. Producer threads must not add more data into the buffer if it's full. The consumers must not read data from the buffer if it's empty. [1]

The problem only requires a rather simple algorithm to solve it but it is already becoming difficult for programmers to really understand all the algorithm's complexities. If we could change the program to work in ready-made thread-safe code-blocks it would be a rather simple algorithm. Examples of a traditional inter-thread communication and a simplified version with multithreading framework are shown in Figures 4.4 and 4.5, respectively.

In the traditional solution the buffer is protected by semaphores. In C++ the storage can be implemented as a class with two methods: `Add` and `Get`. Each method

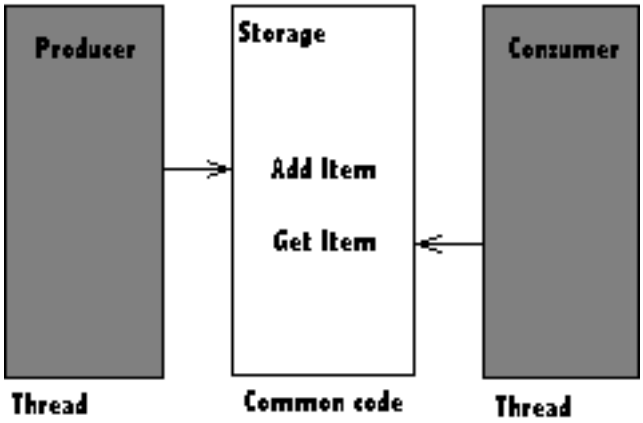


Figure 4.4: Traditional Consumer–Producer implementation

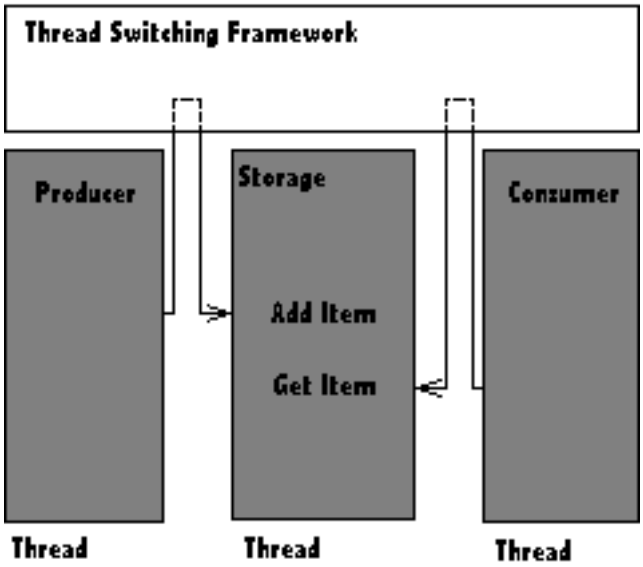


Figure 4.5: Consumer–Producer implemented with the framework

uses the semaphores to protect the shared data, thus allowing an implementation with very little overhead.

With the framework, the implementation is not as light as the traditional one, but it provides some great advantages over the traditional version. With Producer, Storage and Consumer each being run in their own threads, every method can be implemented without explicit protection. When any other thread calls the provided methods through the framework, the call will be relayed to correct thread and the result will be returned in the caller's own thread. There is no risk of corrupting shared data.

Another benefit is that all components work in different threads instead of several threads sharing one buffer. Any component only has access to its own private data. The algorithm is as follows.

1. Any producer-thread registers itself to storage-thread. This is required for informing producers that there is space for adding new items.

2. Producer-threads send a data payload for storage-thread. If the storage becomes full, a return value is given and the producer goes to sleep. If the storage was empty, storage-thread will call all consumers to start up. The return value can be used to cache the payload.

3. Any consumer thread registers itself to storage-thread. This is required for informing the consumers that there are items ready to be removed.

4. Consumer starts reading from the storage. If the storage becomes empty, a return value is given and the consumer goes to sleep. If the storage was full when the data was read, all the producers will be called to start up.

There is also a functional difference compared to the solution without the framework, the fourth phase in the algorithm. See Section 6.2.2 for a more comprehensive explanation why it is needed.

Also, total amount of lines-of-code - as well as the size of the executable binary - required to do this is significantly larger than the one described in the traditional version. Also the efficiency of the application is most likely significantly poorer than that of the optimized algorithm, but then again the algorithm is now on a much higher level: it's easy to read, write and understand.

Writing and testing new optimized multithreaded algorithm for a new problem can take an overwhelming effort. The best thing about the new approach is that the new algorithm does not require any semaphores or mutexes from the newly-created code as all the synchronization is done inside the framework. If there is a problem with the synchronization, it only needs to be fixed once, in the framework, and not for all new applications. When the framework gets maturised, most of the errors would get fixed and all new applications should be that much more robust.

As all threads exist in a single process they have access to each others' data. Therefore any dynamically allocated memory can be accessed by any thread and therefore the integrity of the data may also be compromised. This means that even though there is no need to protect the data in the user code, it is still possible to break integrity by implementing malicious code. However, it is possible to prevent that from happening by using suggested coding methods.

The framework is supposed to alleviate this issue by using `auto_ptr` type smart pointers to pass data from one thread to another. This has two main advantages: firstly, auto_ptr is a reference counting pointer, so the allocated data will always be deleted when it's not used. Secondly, the ownership of the data is always passed with the pointer. In other words, the design and the programming language forces

the programmer to use appropriate datatypes and therefore guides the programmer to understand that after passing the data to the other thread, the calling thread cannot access the data anymore.

Just as the request itself is passed over from thread to another via the auto_ptr, the return value is also passed within the request. See Figure 4.6 for explanation. The framework knows nothing about the return values, it only moves the requests around and the embedded data is passed along. In the figure the auto_ptr is passed from client thread to the serving thread via ThreadQueue. Ownership is passed to the serving thread and it may freely access the arguments as well as set the return value before completing the request. After completion ThreadQueue will move the request back to client thread which may read the returned value.

If the return value will be passed via a normal pointer, it may be set to null when issuing the request. The serving thread can allocate the memory when the request is served. If the request is using pointers as return values, one should consider using smart-pointers to relay them as that will lessen the risk of memory handling problems.

This allows the possibility of passing multiple return values within any request, without adding any overhead. However, the programmer should seriously consider not adding more than one return value as the need for several return values may also be a sign of a design flaw.

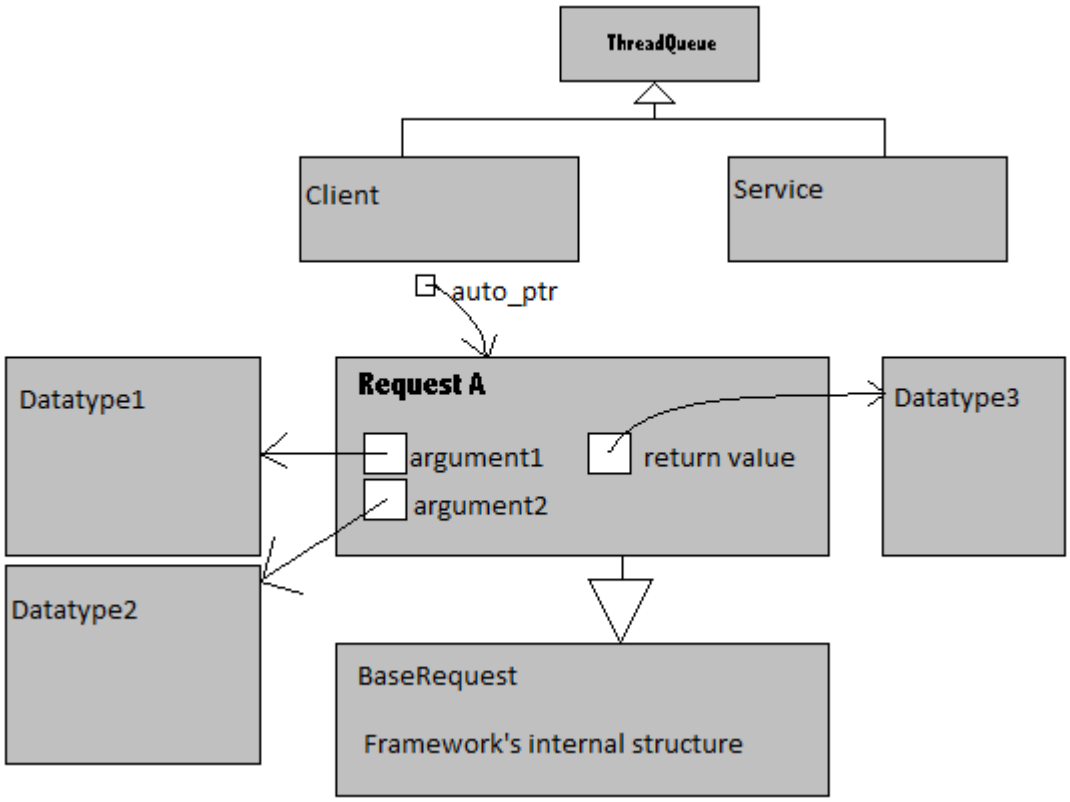


Figure 4.6: Request's return value

As it was pointed out, it is still possible to misuse this feature by creating two pointers to an object and then initialize the auto_ptr using one of the pointers. The second pointer then remains pointing to the object and the data can be accessed by both threads. This is an acceptable risk as the application needs to misbehave deliberately. If the framework's coding convention is followed, there is no such problem.

There is also another keyword about protecting data-integrity in C++ called `volatile`. The `volatile` keyword basically tells the compiler to prevent the specified variable from being cached in the processors registers. In Figure 4.7 there is a multithreaded application running on two cores with a non-volatile variable in memory. Core #2 has read the value of the variable and has altered its value. But, as the variable is not declared volatile, it can be cached by the processor and the altered value does not get updated to the memory. It is possible for core #1 to start reading the value of the variable and it is possible to read the non-altered value – thus leading to data-integrity error. As the framework will always handle data in only one thread, this error should never occur, but if it does, the user code could set the data being sent to other threads as a volatile member variable.

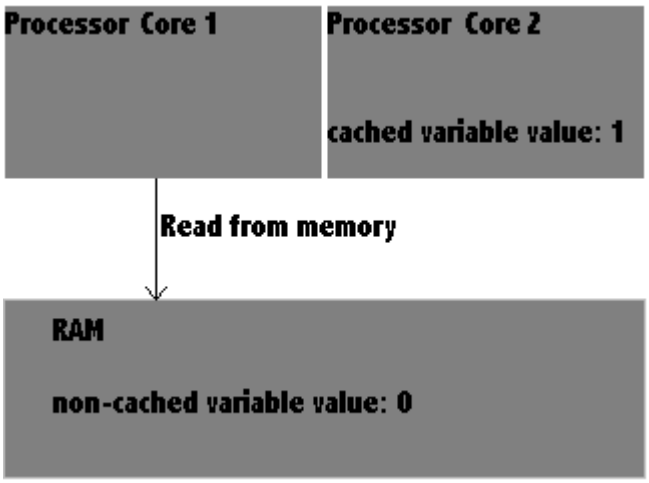


Figure 4.7: Non-volatile member cached

### 4.2.4 Request sending shall return immediately

One aspect of the framework is that all calls to other threads will always return to the calling code as soon as the request is enqueued. This way any call to another thread will never hang. Calls hanging in other components can be extremely difficult to debug as the programmer might not be familiar with the hanging code. Or, in some cases there might not even be availability to the source code of the called thread, only the binary. In order to prevent these kinds of problems, it is best that the

thread switching is done automatically by the framework and it can be quaranteed that the calling thread does not hang.

Another benefit of this approach is that as thread switching can be implemented inside the framework, it allows a possibility for optimisations that make the framework's data protection mutexes to be reserved for only bare minimum of time. Also, most importantly, this assures that the calling thread never executes any of the other thread's code. Using the framework the other thread's data is automatically protected to prevent any other thread calling the user-generated code directly, preventing concurrency problems and making the user-generated code easier to read.

Also, as all the requests will be relayed as a task-based ideology, the framework can keep its stack size reasonably small. The framework will be run in the thread's main-function calling the user-generated code. This means that when user-code is executed, there will be very little stack space allocated, leaving the user-code nearly the maximum amount of stack space available for use.

### 4.2.5 Embedded requests

The framework shall allow tasks to have sub-requests. What this means in practice is that sometimes a thread needs to send a single request that is very complex in nature. Sometimes the request even has clearly separate functionalities embedded in it that should actually be two separate requests.

For example, if one were to implement a device handler, one request might be to turn display on. With some simplification, this request could consist of three different tasks that have little in common, such as power on the display, draw image on the display and finally enable display output.

All this functionality in one request needs a lot of code inside one handler function which increases code complexity. Also, duplication of code is a clear risk as different requests might have separate functions for enabling and disabling resources.

Usually, the first step to approaching an elegant solution to a problem is to divide the problem into several smaller problems, reiterate and finally resolve the smaller problems. Using this approach, any resource handling would consist of one big request that handles the full operation in several smaller requests. Furthermore, any drawing action should have its own request and enabling display output should have its own. All in all, there would be three clearly distinct requests. So, the single "display on" request can be considered as a meta-task that will simply keep adding new tasks into the queue as it continues to be server.

There is still a problem with this approach: after simplification the "display on"-request might have three separate phases: "power up", "draw image", "show image". Also, let's say there is a turn display off request that requires only one step.

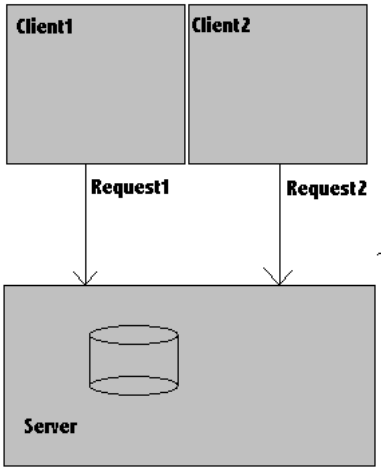


Figure 4.8: Two clients interacting with a common server

Now, if there are two separate clients who are sending these requests, each in their own threads, and then there is the display request server in its own thread. Figure 4.8 shows two separate clients interacting with a server. In the server there is a task queue, which can be thought of as a linked list.

First client1 is sending a request1 that consists of several independent phases. Before request1 has been served, the client2 gets executed and issues "display off" request. Now there will be two requests in the queue, request1 and request2. When request1 is being executed, it will add the embedded request into the queue, thus splitting the request1 into two parts, request1 and request1_2, as shown in Figure 4.9. In order to quarantee correct order of execution, the sequences would have to be appended as shown in 4.10, otherwise the request2 would actually be served before request1 has been served in full. In some situations such an error in sequences might cause a crash or at least data corruption. Unfortunately, without some extra specification to the requests, this kind of rearrangement of requests cannot happen automatically as it cannot be known which slots all new embedded requests should fall into.

So, there is clearly a problem that some requests may become interleaved and then interfere with each others' execution. Without a threaded framework this kind of issue would be dealt with by adding a new semaphore and protecting the sending of the requests. Naturally that cannot be done now because the framework simply


Figure 4.9: Embedded requests in invalid ordering


Figure 4.10: Embedded requests in intended ordering

takes requests as they come in and then serve them, and that is how it should be: keep it simple. So, clearly the "display on" needs to be an own request as a whole, but then we are back where we started.

The embedded requests, or subrequests, are one way of minimizing the problems caused by scenarios such as mentioned above. In order to keep the "master"-request simple, all the parts of the request should be split – just like was done above – and then the master release creates one new request of each and adds them to itself as subrequests.

Adding subrequest works exactly like sending any request to another thread, except that the current request will only be completed after all its subrequests are served and ready. Also, as a subrequest is nothing but a normal request, it too can have subrequests. This is called the composite pattern [6].

Figure 4.11 shows an example scenario where request A has had a subrequest added to itself. This means that until Request C is complete, request B will not be executed. But as subrequests are also merely normal requests, request C can also have a number of subrequests added to it, which means that C will only be completed when all its subrequests have been finished. Subrequest D will be executed and completed first, after which the execution is passed to request E. When E is done, it will be completed and request C may complete itself. After completion of C, A can also complete and pass execution to request B.

Requests may never be added in front of a request that has already been queued, as that would endanger the integrity of the request structure.

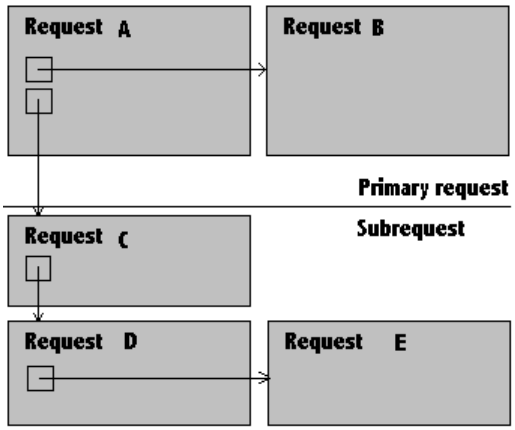


Figure 4.11: Request-subrequest relations

### 4.2.6 Differentiating of signals and callbacks

For simplicity's sake it may be necessary for some requests to not have a callback at all. From here on, requests without a callback will be referred to as signals.

If a request has a callback, it means that after completion of the request, the framework will automatically issue a callback into the request's sender's thread, more specifically the same request handler that sent the request. In conventional callback ideology, the callback in the caller's code will be executed in the serving thread. This creates a severe risk that may cause deadlocks or data corruption, as was mentioned already in Section 3.2. Also, the thread that calls the callback can never be sure of what the client-side code will perform in the calling thread, so both the caller and the serving thread are both at risk of executing some arbitrary code.

One way to ensure safe callbacks is to have an agreement that the only thing that the callback does is to signal the other thread about the callback. In C++ this could mean unlocking mutex or semaphore.

In, for example, Symbian this kind of problem has been circumvented by sending TRequestStatus objects which are completed by the serving thread and waited by the calling thread. The culprit of a mere agreement, or coding convention, is that it is impossible to force correct behavior, as the implementation might simply not follow the agreement. Potentially malicious code could be run in the calling thread. [16]

The framework shall implicitly ensure that all callbacks are executed within correct threads. The request-serving thread can safely perform the callback as it is known that the callback will always enter the user-generated code in the client's own thread. This way, if there is any malicious or unstable code in the callback, the serving thread is not in danger.

See Figure 4.12 for a visual representation of how callbacks shall work. The CallerThread object starts in a Handler-method case 1, issues a request (1.) with a callback to the ServerThread. When the request has been served (2.) by ServerThread it will exit the Handler()-method and the framework will automatically detect that there are no more embedded requests unserved, as well as no more callbacks are expected from other threads, so it will automatically set the request as Completed (3.). When `ThreadQueue` detects a completed request it will check whether there was a callback pointer set, and as it was in this case, it will call the handler for the callback in the calling thread. Thus, CallerThread will reactivate and perform a call in it's own context and call the Handler. User code can again use an iPhase integer member that should be incremented prior to calling the handler, so it is possible to divide the behavior into a clearly defined switch-case structure.

A conventional asynchronous request is called in the user-thread with a pointer to a callback function as a parameter. A graphical representation of the calls can be

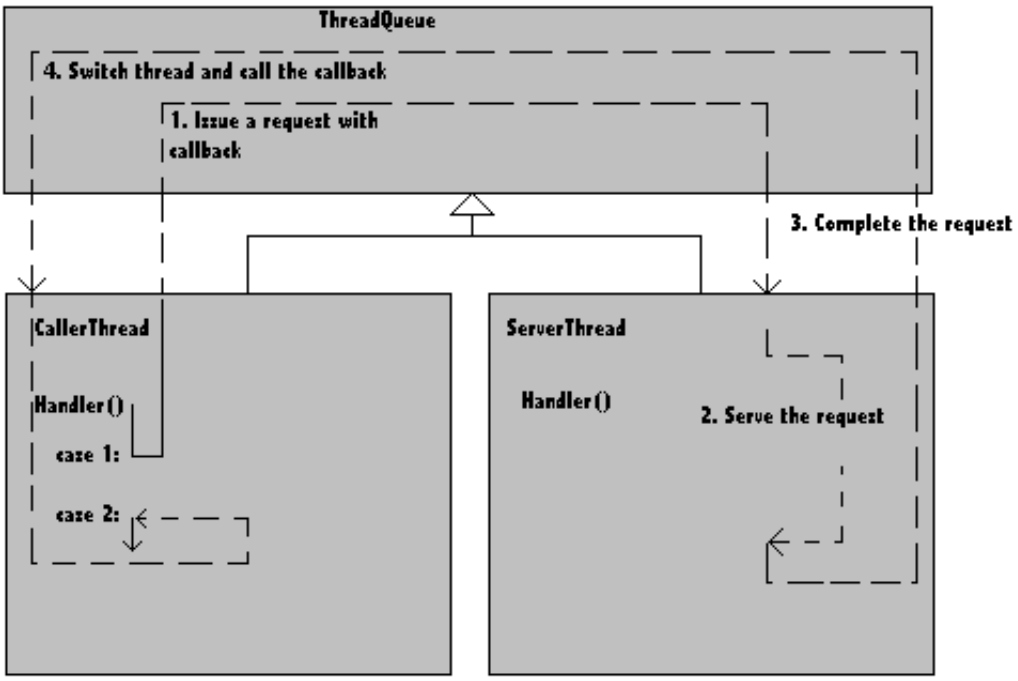


Figure 4.12: Callback phases

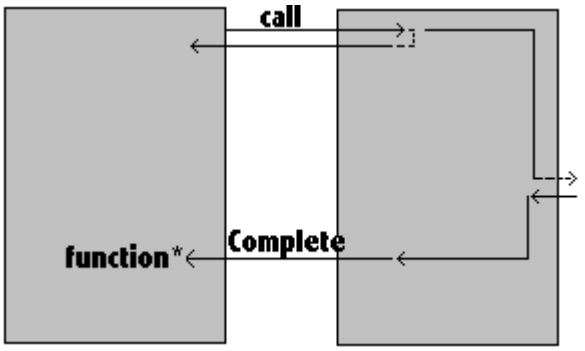


Figure 4.13: Conventional callback

seen in Figure 4.13. Although this kind of callback structure is greatly more simple, this kind of conventional callback is not thread safe by itself, as the developer must explicitly take care of securing data integrity.

### 4.2.7 Simultaneous requests

Any thread shall have the ability to call multiple requests with a callback in a row before waiting for any callbacks. This way the source thread can make use of the waiting time of the asynchronous requests without having to wait idle. It is extremely important to note that when calling multiple requests at the same time, the first sent request will receive its callback first. This means that when reading the callback value from the request it will be the return value of the first request. This is important because reading a return value requires casting for the object that

is returned from the framework. This is one of the most dangerous drawbacks with the framework, but one can live with it when programmers obey the rules set by the API.

Sending requests with callbacks also pose another problem. When a thread sends a request with a callback, the source thread will ultimately wait for the request to be complete. Until the request is completed via a callback the thread will not start serving any other requests. This means that the sent request must not cause a circular stream of callback-requests that end up sending a callback-request to the waiting thread. This will cause a deadlock for all the threads and there is no way to recover from this situation. If and when such situation occurs, the user of the framework shall change the design of the program so that the circular behavior does not occur.

This kind of faulty design is shown in Figure 4.14 on the left. Component A calls B with a callback. Then – within the request – component B calls component C with a callback, which also calls component A with a callback. Component A will never start serving the C's request as it's waiting for B's callback and C will never callback to B. If the callback-request must call the originator of the request, it can only do so with a non-callback-request. This will work as a signal between the threads and it will not hang the system. In other words, if the call from C to A does not expect a callback, this design will work.

If the programmer follows the advice from Stroustrup's book [15], and designs the thread structure to form a tree, there will be no problems with deadlocks. Example of the desired approach is shown in Figure 4.14 on the right.

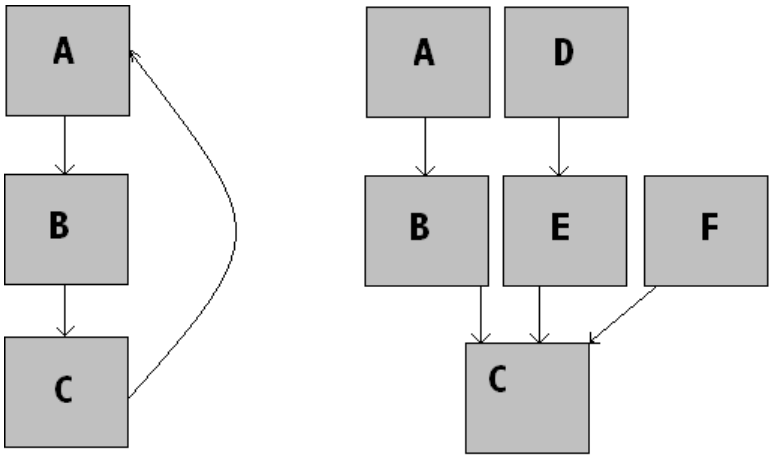


Figure 4.14: Faulty and preferred designs

# 5. MULTITHREADING FRAMEWORK USAGE

## 5.1 BaseRequest

First API is the `BaseRequest` header file. The role of BaseRequest object is to provide users of the framework a simple way to create customized requests that can be passed to other threads. `BaseRequest` itself holds some important information that allows the framework to work, such as pointer to the calling thread, list of subrequests and information if the request should be completed with a callback. `BaseRequest` basically has all the information required by the framework and the implementing class will only have all the information required by the application, such as parameters and return values.

BaseRequest class is a simple interface that only has constructor, virtual destructor and a pure virtual `act` method. The constructor can take only one parameter, and that is only required when the request should send a callback. The callback-parameter has to be a pointer to `ThreadQueue` which is creating the request. As the callback is a pointer to a dynamically bound object, there is no need to implement static functions to which function pointers can be created. See Figure 5.1 for a class diagram. The framework internals use classes derived from SimpleListItem to track the status of the requests. Although they are not visible to users in any way, they cause unfortunate overhead when they need to be allocated and deleted. Caching pre-allocated items would decrease the overhead, but it cannot be removed completely.

When issuing requests, `ThreadQueue` class will automatically add the pointer into an embedded structure that is passed within the request to the other thread. That pointer will be used to signal a mutex after the request has been completed. In the sender's thread the framework will lock that same mutex right after calling `Act()` virtual method. If the sent request was a signal, without callback address, the mutex locking will not be done. After the serving thread signals the mutex, the sender thread is freed and it will again call `Act()`. This loop will be continued as long as new requests with callbacks are issued. After there are no more callbacks left, `ThreadQueue` will check **that** request's embedded information about callbacks, and if there is a callback address, this thread will also signal a mutex. This kind of never-ending stream of requests will keep the program alive and in synch.

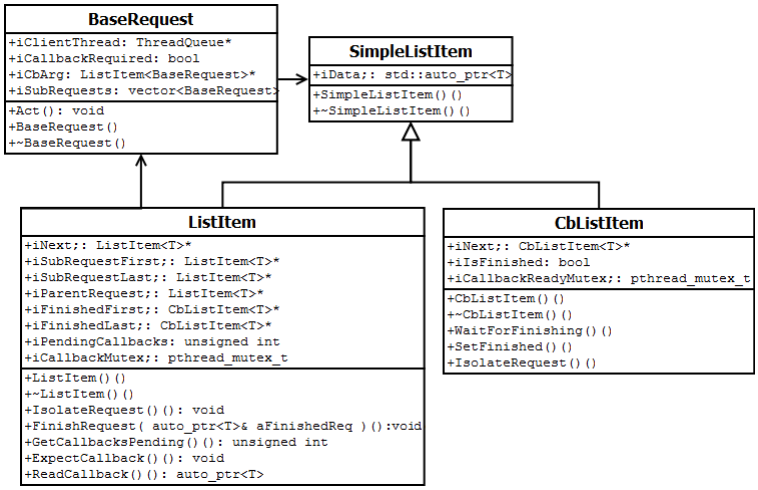


Figure 5.1: BaseRequest class diagram

This makes it very simple to define which request actually has a callback and which request is simply a signal. A signalling request could be used in, for example, logging services, where the serving thread is not interested in when the logger has actually printed the text.

On the other hand, any multiphased requests can easily be implemented with the callback. Thread can create a phase structure into the request handler, in which the first phase creates a new request that is to be sent to the other thread, send the request and simply return from the function. When the other server has completed serving the request, the framework will automatically call the same request handling function with the phase number incremented.

Also, the request must have a virtual method called `Act()`. This function will be called by the framework in the request-serving thread when it's time for this request to be served in the serving thread. Usually the `Act()` method will call the `ThreadQueue`-inherited object's request handler method. This allows for very minimalistic amount of work requirement for creating a request that should be served by the other thread.

The `Act()` method is set as a public pure virtual function, but it does not mean that it should be called by user code. In fact, it should never be explicitly called by any other component but the framework. Optimally, the `BaseRequest` would be set as a friend-class of `ThreadQueue` and vice versa, which could allow the method to be set to private namespace. That way there would not even be a possibility of calling it incorrectly.

## 5.2 ThreadQueue

The second, but most important API in the framework is the ThreadQueue header. See Figure 5.2 for the class diagram of ThreadQueue.

`ThreadQueue` class is meant to be a base class for all new threads. This header is also rather simple to use: in order to create a new thread, the programmer inherits from `ThreadQueue` class. Then, after allocating an instance of the thread, user has to call `Run()` to start the thread. Rest of the functionality is left to the framework.

For users of the API there are five really important methods. First is the constructor which takes a string parameter for naming the thread. Name of the thread is meant only for debugging purposes, it has nothing to do with functionality, but it can be helpful when trying to find errors.

Another basic method is the virtual destructor. There is not much special about that method as it will clean up the thread when it is being deleted. Only the owner of the thread may call `Kill()` to the thread before deleting the object. `Kill()` works in a synchronous manner, so the thread could be terminated when call to `Kill()` returns.

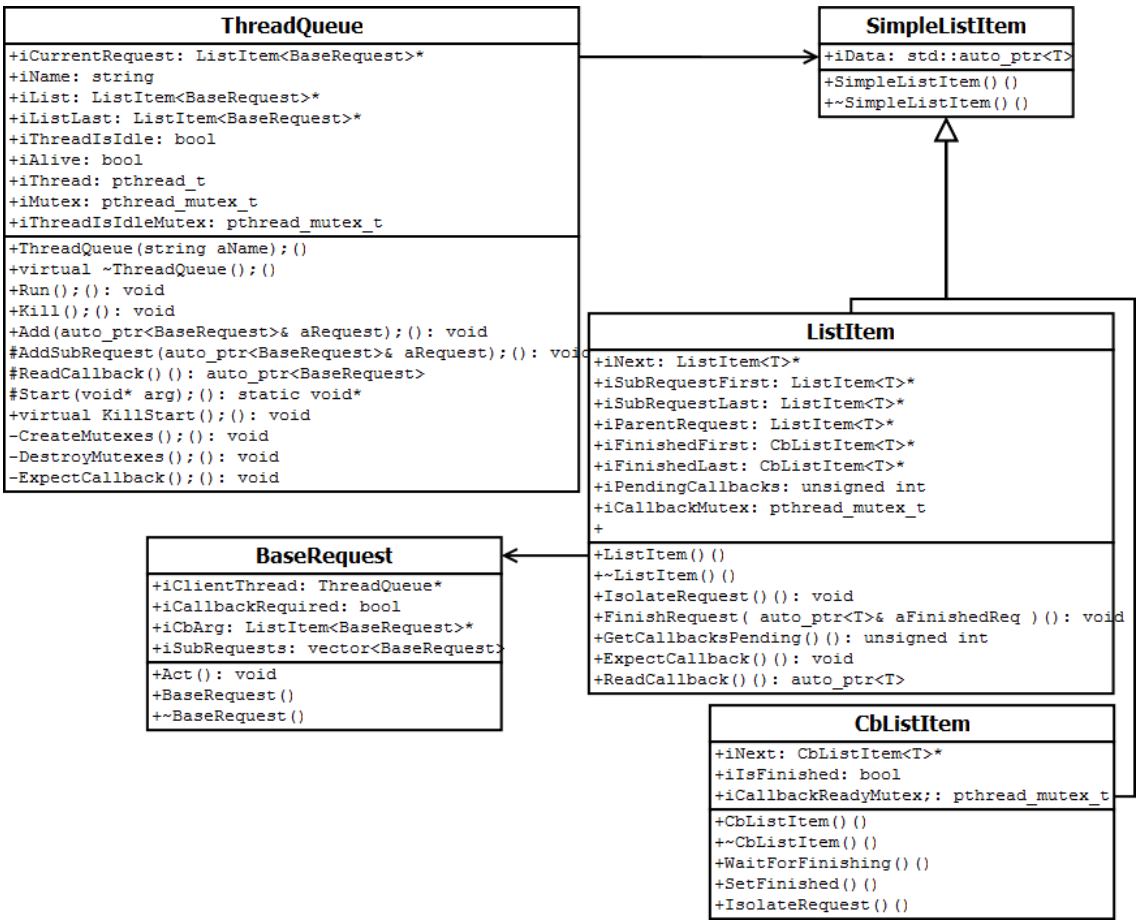


Figure 5.2: ThreadQueue class diagram

After creating a new thread the owner shall call `Run()` in order to start the thread. This method cannot be combined to the constructor as otherwise there is a risk that a thread will call a virtual function before initializing the inherited parts of the object. So, it is imperative that the `Run()` is performed separately after the

constructor is finished. If the constructor throws an exception, calling `Run()` may cause undefined behavior.

When the thread is running, it can be stopped by calling `Kill()`. `Run()` must be called before calling `Kill()`, doing otherwise will cause undefined behavior. Also, in the current version of the framework it must be noted that killing a thread with outstanding requests can lead to undefined behavior. So, before calling `Kill()` the system must be in a state where there are no more requests to be served. Adding functionality that allows `Kill()` to be called at any time is left for future versions of the framework.

Then, finally, there is the method that allows the most important aspect: inter-thread communication. The method `Add()` is used to for sending requests to the target thread. As the parameter-type suggests, all the requests must be inherited from `BaseRequest`. Calling `Add()` will always return as soon as the new request has been appended into the target thread's request-queue. Sending a request without a callback is called signaling, as the source-thread will never wait for any synchronization. Any request that has been sent with a callback will cause the source-thread to call `BaseRequest::Act()` pure virtual method as soon as the request is completed.

Sending a signal request can be extremely useful, and simple, when programmers want to have logging from multiple threads. A single logger-thread can be used to output all the sent messages to `std::cout` or `std::err` and the messages will never be malformed as they're always served by one thread only. It should be noted that attempting to write to `std::` streams from multiple threads simultaneously will end up sending malformed output into the stream.

## 5.3 Framework Internals

The inner workings of `ThreadQueue` and `BaseRequest` are rather complicated, so only the most basic functionality on a high level is shown in Figure 5.3. In the sequence there is a main program which creates two separate threads. Each thread is simply a class which is derived from `ThreadQueue`. One of the derived classes is illustrated as UserThread, the other thread's user code is not shown in the image.

After creation `UserThread` creates a new inter-thread request with pointer to itself as initialization parameter. The inter-thread request is derived from `BaseRequest`. `UserThread` then initializes the arguments and marks return value pointer as null. After that it calls the other thread's Add() method, which takes the `auto_ptr<BaseRequest>` as a parameter.

The serving thread takes this pointer and adds it into its private member `iList`. Because the BaseRequest was constructed with a pointer to a valid thread as a parameter it assumes that a callback shall be issued after completion. Therefore the

other thread goes ahead and increments `iPendingCallbacks`. Add method returns and the serving thread is now idle. `UserThread`'s `ThreadQueue` begins to wait for a callback.

Serving thread becomes alive as it notices a new request in queue. `ThreadQueue` picks the first request from the queue and calls its `Act()` pure virtual method. The request handler is called and the request is serviced in the user code. After servicing, user code sets the return value and simply returns. As this `ListItem` does not have any pending callbacks it will consider this request to be served. Serving thread calls the callback of the `UserThread`, which will call it's own request handler.

Serving thread will now clean up and delete the `ListItem` it does not need any more. `UserThread`'s user code returns and the thread moves to idle state.

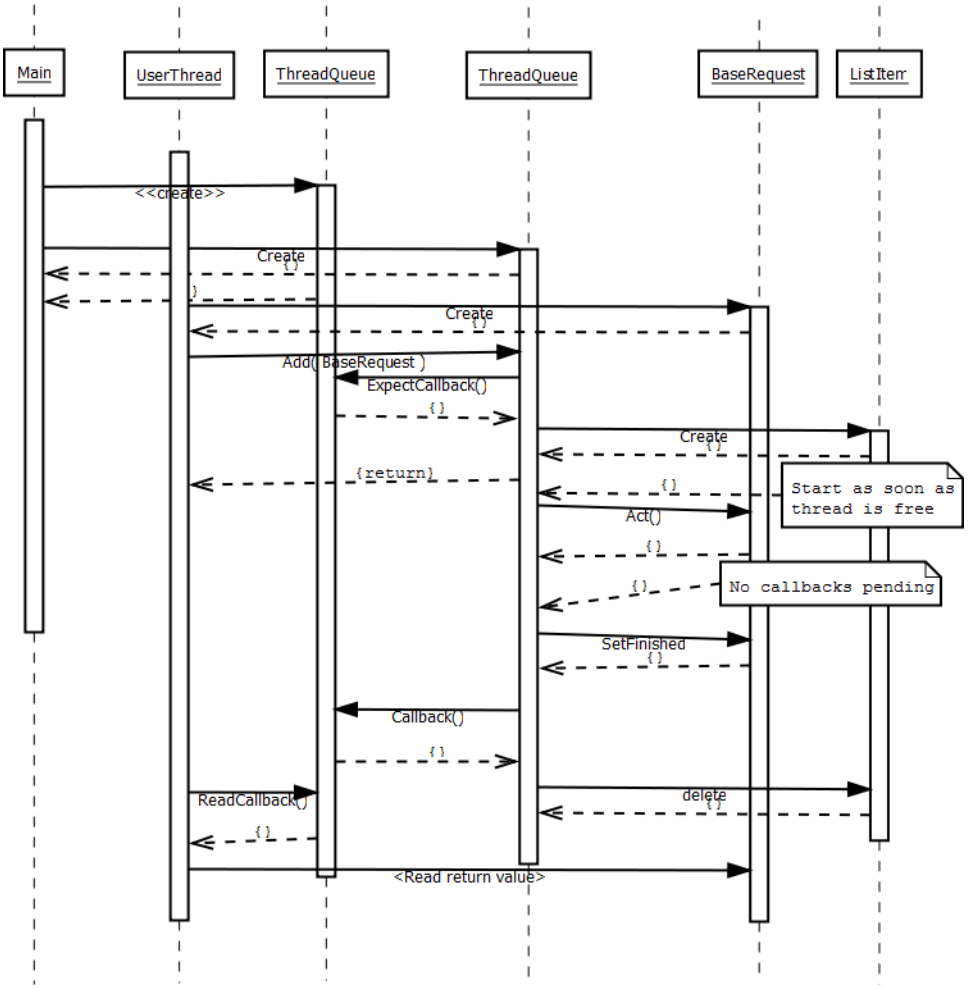


Figure 5.3: Inter-thread request sequence

This functionality can be much deeper when serving threads also issue requests to other threads. Also, when requests have sub-requests in them, there can be, for example, a dozen `Act()` calls from `ThreadQueue` to `BaseRequest` until one request is fully served and the callback is issued.

## 5.4 Suggested coding conventions

### 5.4.1 Source files

In order to keep the code readable, the programmer is adviced to create all the objects so that all the code that will be run in one thread is placed in one place. In practice this means that one object that inherits from `ThreadQueue` will be able to serve several different requests. All these requests should have their code placed either into the same file as the inherited thread, or at least into a separate directory. This way, it will be clearer for programmers to understand in which thread the code will be run.

The return values pose a slight deviation to this rule, as the client thread will need to access them after getting the callback. As the framework is passing the request which holds the return value via an auto_ptr, it is always safe to access the data.

### 5.4.2 Request handlers

As for creation of threads and requests, it should be mentioned that the intention for user-code implementation is as follows. All `BaseRequest` inherited objects have an `Act()` pure virtual method which will be called by `ThreadQueue`. In order to allow easy modularisation of threads, it is recommended that all requests hold a pointer to the target `ThreadQueue` and that thread's request-handler is called in `Act()`. This should make all requests small in size and make it easy to hold all application logic in `ThreadQueue` request handlers. However, this is just a recommended coding convention and the framework does not pose any restriction to this, so any other approach users see fit, may be used.

The amount of code in each handler is dependent on the logical size of the request, but a good rule of thumb has been to limit the number of states in request handler to less than five. Following that guideline, it's quite easy to break down the requests into smaller and more easily understandable blocks.

### 5.4.3 Entry point synchronization

Another interesting point to contemplate is how to manage the entry point of the thread. Let's say a thread is started from void `Run(void* aPtr)`. In Figure 5.4 is a typical entry point for a thread. The example code implements an eternal loop, which will call two independent functions and synchronize after each one is completed. It does not matter which execution takes longer, as the `mutex->lock()` will always halt the executing thread until call to foo is completed. As `bar` is a synchronous call, it cannot return until it is complete. A visual representation can

```
void Run(void* aPtr)
  {
  // Pointer to an object
  MyThread thisThread* =
    reinterpret_cast<MyThread*>( aPtr );

  while( 1 ) // eternal loop
    {
    // call another thread with a synchronization
    // primitive as a parameter
    thisThread->iThread->foo( thisThread->iMutex );

    // execute independent code in this thread
    thisThread->bar();

    // Synchronize, the mutex shall be
    // released the other thread.
    thisThread->iMutex->lock();
    }
  }
```

Figure 5.4: Typical thread's entry point

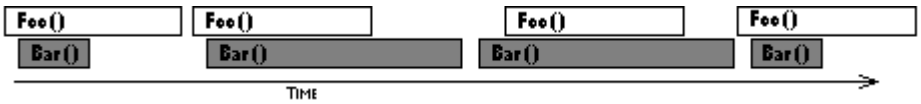


Figure 5.5: Visualization of entry point synchronization

be seen in Figure 5.5, where it is graphically shown how the synchronization happens. `Bar` is shown to start before and after foo in order to illustrate the fact that both the call to foo may be simply a signal and might be queued, thus not starting to execute immediately. However, another call to foo cannot start until the previous call to `Bar` is finished.

Inside the while-loop, there is a mutex locking, because usually threads require some sort of wait or synchronization before they start a new run, be it a video-player that runs exactly at 30 frames per second, or a keyboard-handler that awaits for a key-press.

If the thread does not use a mutex to put the thread to sleep, it would have to wait in a busy-loop until it is intended to continue. Busy loops are appropriate only in very low-level programs, where a wait call to operating system could actually force a longer wait than needed. For example, if the operating system supports blocking wait function for microseconds but the program running only needs to wait for nanoseconds.

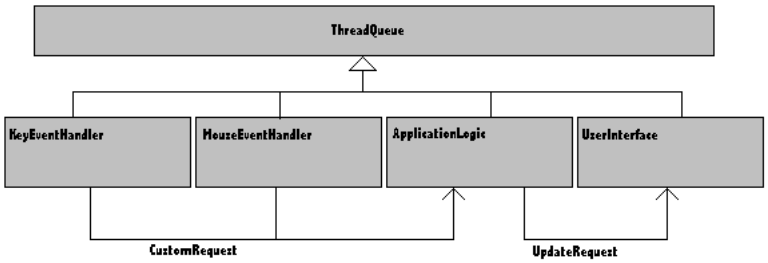


Figure 5.6: Event handler using framework

By using the framework the user could simply exit from the running function and the framework will put the thread to sleep automatically. Then, when an event happens that requires this thread to run this thread will be called with an appropriate request. An example of this behavior could be a keyboard event handler. See Figure 5.6 for a class diagram and Figure 5.7 for short code example of such application that uses the framework. The `KeyHandler` simply reads user input and sends signals to `ApplicationLogic`, which can act on the input.

```
void KeyEventHandler::Handler()
{
  while( 1 ) // eternal loop
    {
    Std::string str;
    std::cin >> str;
    iApplicationLogic->Event( str );
    }
}

void ApplicationLogic::Event(std::string aStr)
{
  // create a new request and pass it to framework
  // no second parameter, so no callback
  KeyRequest* req = new KeyRequest( this );
  // pass the input to the logichandler
  req->iPayload = aStr;
  auto_ptr<BaseRequest> reqP( req );
  // make auto_ptr the only reference to the request
  req = NULL;
  // send to framework, returns immediately
  Add( reqP );
  return;
}

void ApplicationLogic::Handler(KeyRequest* aReq)
{
  // use the request here through the pointer
}
```

Figure 5.7: KeyHandler example code

### 5.4.4 Exception throwing

In order to make the framework simpler, the applications cannot use try-catch blocks to catch exceptions that happen within a request serving. When the request is sent to another thread the control returns immediately after the request is enqueued, so the actual execution of the request serving is not included into the try-catch block. So, it is reasonable to minimize overhead by not using try-catch in request sending.

Also, due to the aforementioned reason, it is not allowed to throw an expection from a request handler. If there is a throw statement during the execution of the request, it will be caught and handled by the framework. By default the framework's error-handling is that the program will abort.

# 6. EVALUATION

## 6.1 Requirement evaluation

In order to be able to assess the successfulness of the work, all the requirements set for the framework should be evaluated.

In Requirement 1 it is required that code written using the framework shall be simpler than traditional multithreaded code and it shall allow easy extensions to APIs. Both cases are considered to be passed. Still, the coding conventions in Section 5.4 should be referred to in order to have consistency in the code.

Requirement 2 states that mutex-handling should be minimized. Using request sending via the framework, developers do not have to use any mutexes, so this requirement is clearly passed.

Shared data protection in Requirement 3 is a much more difficult requirement. Firstly, the framework allows easy creation of shared data storages that are thread safe, so in that sense the requirement is passed. Still, it is possible to pass pointers via the framework that can be used to corrupt data. But as the framework forces the use of auto_ptr and suggested coding convention state that the developer must not access data from two threads simultaneously, the requirement can be considered as a pass.

Requirement 4 states that when a thread sends a request to another thread, the sending thread must not execute the request, ever. Instead execution shall immediately return back. As the user of the framework cannot accidentally execute the code in a wrong thread, this requirement is also passed.

Next, Requirement 5 defines sub-requests and how they should work. As the framework allows embedding requests within each other, and as they work as defined, this requirement is passed.

The framework differentiates signals and callbacks as required in Requirement 6. When BaseRequest is constructed with a pointer as a parameter that pointer will be used as a callback. This requirement is implemented a bit poorly, as the user has to explicitly pass the `this` pointer of the executing ThreadQueue, otherwise the program might fault. This is currently not implemented so that compiler could force correct behavior. On the other hand, the requirement only states that signals must not call a callback and other requests shall call a callback, and that is implemented, therefore this requirement is also considered as a pass.

The final requirement, Requirement 7, was to allow simultaneous requests. This requirement is actually really hard to implement well. Currently it poses lots of restrictions for the user of the framework in the sense that developers must know their application well so that no deadlocks occur due to circular request sending. Another, more serious restriction is that user must read the return values in a specific order. This causes that if the first request takes longest to finish, the other return values cannot be read until the first one is finished. The implementation is not sound, it is even error prone, but it works if it is used as is required. But due to the restrictions, this requirement is only partially passed.

As a whole, the requirements are mostly achieved, although some of them had disadvantages in them, and the implementation can be regarded a success.

## 6.2 Producer–consumer comparison

Performance of a framework is often considered to be one of its most important aspects. For this work however, the approach is that performance is not even in the requirements. But since performance is considered one of the most important factors in programming, it should be analyzed also.

An algorithm that is already somewhat efficient was chosen to the performance analysis. The producer-consumer problem is an old basic example of a concurrent programming error, so it suits well to the analysis section. A reference implementation was searched from literature, and as one was presented by Järvinen and Haikala [7] it was chosen. Unfortunately it was implemented in Ada, so it had to be ported to C++ for better comparison.

In both implementations the payload carried to Consumer will consist of a dynamically allocated string initialized with C-string "`textit`". That is 48 bytes of data being allocated, passed via a pointer and deleted.

Performance comparison runs were run on an AMD dualcore system, running at 2.50GHz. The system has 4GB of memory and approximately 1TB of hard drive. The operating system is Windows XP Professional with Windows Service Pack 3.

Framework was implemented first with Apple XCode IDE, but during the debugging phase the program was ported to Windows, at which point the programming and debugging continued with Microsoft Visual C++ 2005 Express Edition. The binaries for performance comparison was compiled with Visual C++ Express Edition's default compiler.

Pthreads library is linked against pthreadVCE2.lib which uses pthreadVCE2.dll. This version of the library supports exception throwing. [9]

### 6.2.1 Traditional implementation

The reference-implementation is very simple as it only has two threads and one common storage. Class diagram is shown in Figure 6.1. The storage has two functions and they both have a mutex and semaphore protection for the stored data, so that there will not be overflows or underflows and the integrity of the list is never compromised.

The producer and consumer are both extremely simple, they run in an eternal loop, allocating and deleting memory for new string objects. The most interesting part is naturally the buffer, as all of the semaphore protection is done there.

In order to mimic Ada's accept-when –mechanism, the storage has one mutex to protect the buffer. It also needs to have one semaphore with a starting value of maximum buffer size (iFullMutex) and one semaphore with a starting value of 0 (iEmptyMutex).

Whenever a new object is put into the buffer iEmptyMutex is signaled once and iFullMutex is waited once. Likewise, whenever an object is extracted from the buffer iEmptyMutex is waited once and iFullMutex is signaled once. This way any number of threads that may attempt to put an object into a full buffer, or get an object from an empty buffer will go to sleep until the buffer is ready for it.

All the data is passed via pointers in order to get maximum efficiency.

### 6.2.2 Implementation using the framework

The new implementation will use the multithreading framework. See Figure 6.2 for the class diagram of the impelementation. It can immediately be seen that the structure of the implementation using the framework is many times more complicated than the reference implementation.

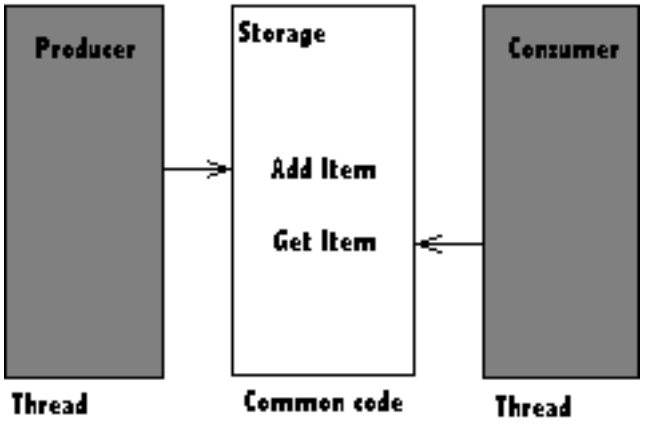


Figure 6.1: Reference implementation

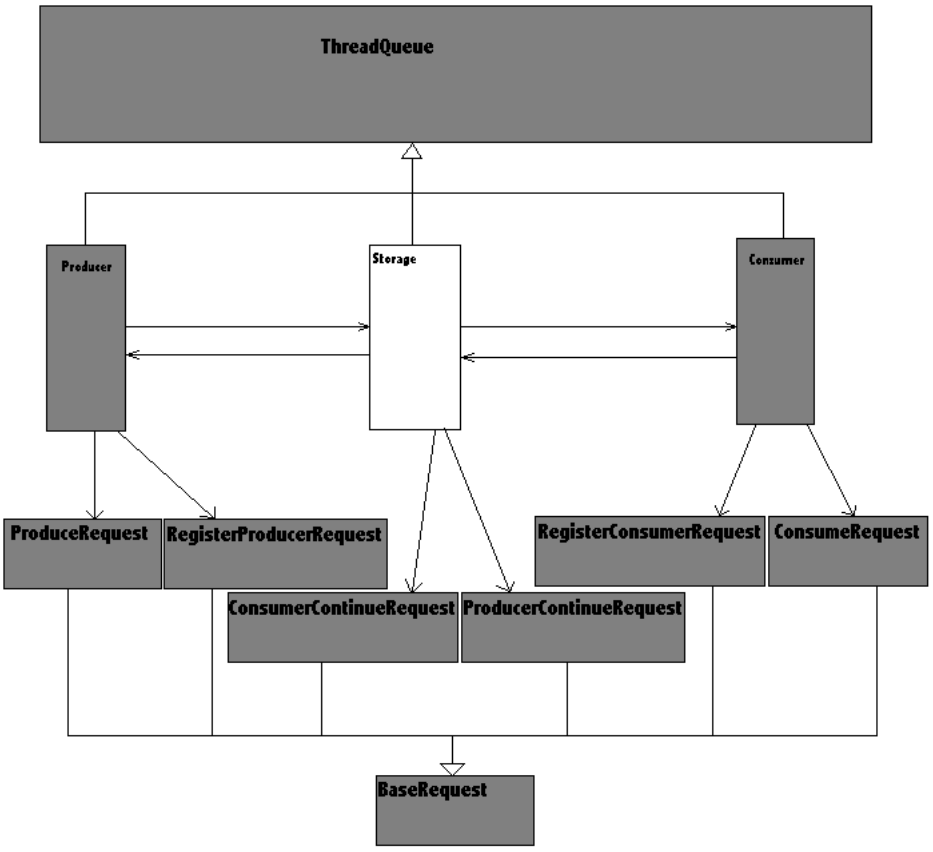


Figure 6.2: Producer-consumer using the framework

However, the design of the framework allows the programmer to completely avoid using semaphores and mutexes, thus making the code itself extremely easy to understand. There are no complex protection of any member anywhere, all implemetantion is actually as simple as a single-threaded implementation.

However, this example proved to be a very interesting one for the framework's task-based design. Sending only requests creates another challenge for efficiently keeping the buffer full. It is not possible for the producer-thread to simply lock a semaphore in the storage-code as storage-code will only be executed in storage's thread. So, a simple semaphore with the starting value of the buffer cannot be used to protect over- and underflow from happening. Although, over- and underflowing does not need a synchronization primitive at all, since the storage can simply check its private member and determine whether there are items in the buffer.

However, a resolution is still needed to prevent producer and consumer from polling the storage. One way to achieve this is to make the storage aware of the producer and consumer threads by registering them to the storage. Then, whenever the buffer becomes empty or full, the storage will need to explicitly signal the producers or the consumers to wake up. This does cause a lot of overhead if the buffer is often empty or full, so it needs to be taken into account. There could be

some ways to use heuristics to wake up only some threads at a time and that way improve efficiency, but as the framework's primary function is to make the code simpler, this kind of performance hit is considered acceptable for now. Perhaps it is even desired as it shows the kind of problems that arise from changing coding patterns.

### 6.2.3 Results

The performance problem that was mentioned in Section 6.2.2 causes a lot of uncertainty about the efficiency of the implementation with the framework when there is only a small buffer available, so the tests have been run with different buffer sizes. See Tables 6.1 and 6.2 for the tables that show the measured spent times from the two implementations. Figure 6.3 shows a graphical representation of times spent between the two implementations using the same inputs and varied buffer sizes.

Not surprisingly the reference implementation is showing almost completely linear results. It was definitely expected as there is almost insignificant overhead from buffer being full. Therefore, even with extremely large input sizes there will not be extra overhead.

On the other hand, when looking at the results when the framework is being used, it seems to remain nearly linear until some point, see Figures 6.4 and 6.5 for a closer view of spent times. The axis to the left shows the size of the buffer used, axis to the right shows size of input in words, each Bytes long. Interestingly, different buffer sizes do not seem to have much difference when processing data sizes less than half a million words. When input size is grown further, the spent time is also beginning to grow faster and faster.

With less than 100 000 words being sent, both versions appear to be quite close to each other's efficiency. Of course, this is partially an illusion caused by measuring inaccuracies, but still they are rather close in efficiency.

When growing input size to more than 100 000, but less than 1 000 000 words, the framework is beginning to show the extra overhead caused by all extra copying of structures in `ThreadQueue` component compared to the minimalistic lock and unlock of mutex and semaphore with the reference impelentation.

After growing input to more than 1 000 000 words and beyond, the two implementations cannot even compare any more. With 5 000 000 words input the framework is showing ten times more time spent and beyond that the ratio would grow even higher.

So, it must be said that this kind of framework does not suit the needs of any program that requires heavy sending of data from one thread to another, due to the extremely high overhead in large throughput case.

But then again, if programs only need to use threads to synchronize some behavior, this kind of framework could easily provide enough efficiency. For example, if any kind of user input would trigger a multithreaded sequence, such as key-press printing a letter on screen, it would make all the framework's delays look insignificant. So, basically it's all about putting the efficiency into right proportion compared to the needs.

| input [words] | Buffer [words] | Time [s] |
|---:|---:|---:|
| 10 000 | 10 | 0 |
| 100 000 | 10 | 1 |
| 200 000 | 10 | 1 |
| 300 000 | 10 | 2 |
| 400 000 | 10 | 3 |
| 1 000 000 | 10 | 7 |
| 5 000 000 | 10 | 34 |
| 10 000 | 50 | 0 |
| 100 000 | 50 | 1 |
| 200 000 | 50 | 2 |
| 300 000 | 50 | 2 |
| 400 000 | 50 | 2 |
| 1 000 000 | 50 | 7 |
| 5 000 000 | 50 | 33 |
| 10 000 | 100 | 0 |
| 100 000 | 100 | 2 |
| 200 000 | 100 | 1 |
| 300 000 | 100 | 2 |
| 400 000 | 100 | 3 |
| 1 000 000 | 100 | 6 |
| 5 000 000 | 100 | 31 |
| 10 000 | 1 000 | 0 |
| 100 000 | 1 000 | 1 |
| 200 000 | 1 000 | 2 |
| 300 000 | 1 000 | 2 |
| 400 000 | 1 000 | 2 |
| 1 000 000 | 1 000 | 6 |
| 5 000 000 | 1 000 | 29 |
| 10 000 | 10 000 | 1 |
| 100 000 | 10 000 | 0 |
| 200 000 | 10 000 | 1 |
| 300 000 | 10 000 | 2 |
| 400 000 | 10 000 | 3 |
| 1 000 000 | 10 000 | 6 |
| 5 000 000 | 10 000 | 30 |
| 10 000 | 100 000 | 0 |
| 100 000 | 100 000 | 1 |
| 200 000 | 100 000 | 1 |
| 300 000 | 100 000 | 1 |
| 400 000 | 100 000 | 2 |
| 1 000 000 | 100 000 | 6 |
| 5 000 000 | 100 000 | 31 |

Table 6.1: Reference implementation's consumed times

| input [words] | Buffer [words] | Time [s] |
|---:|---:|---:|
| 10 000 | 10 | 1 |
| 100 000 | 10 | 6 |
| 200 000 | 10 | 11 |
| 300 000 | 10 | 17 |
| 400 000 | 10 | 23 |
| 1 000 000 | 10 | 54 |
| 5 000 000 | 10 | 288 |
| 10 000 | 50 | 1 |
| 100 000 | 50 | 6 |
| 200 000 | 50 | 11 |
| 300 000 | 50 | 17 |
| 400 000 | 50 | 20 |
| 1 000 000 | 50 | 54 |
| 5 000 000 | 50 | 272 |
| 10 000 | 100 | 0 |
| 100 000 | 100 | 5 |
| 200 000 | 100 | 11 |
| 300 000 | 100 | 15 |
| 400 000 | 100 | 20 |
| 1 000 000 | 100 | 53 |
| 5 000 000 | 100 | 264 |
| 10 000 | 1 000 | 1 |
| 100 000 | 1 000 | 5 |
| 200 000 | 1 000 | 10 |
| 300 000 | 1 000 | 16 |
| 400 000 | 1 000 | 20 |
| 1 000 000 | 1 000 | 47 |
| 5 000 000 | 1 000 | 242 |
| 10 000 | 10 000 | 1 |
| 100 000 | 10 000 | 5 |
| 200 000 | 10 000 | 10 |
| 300 000 | 10 000 | 14 |
| 400 000 | 10 000 | 19 |
| 1 000 000 | 10 000 | 47 |
| 5 000 000 | 10 000 | 232 |
| 10 000 | 100 000 | 1 |
| 100 000 | 100 000 | 5 |
| 200 000 | 100 000 | 10 |
| 300 000 | 100 000 | 14 |
| 400 000 | 100 000 | 19 |
| 1 000 000 | 100 000 | 46 |
| 5 000 000 | 100 000 | 244 |

Table 6.2: Framework implementation's consumed times

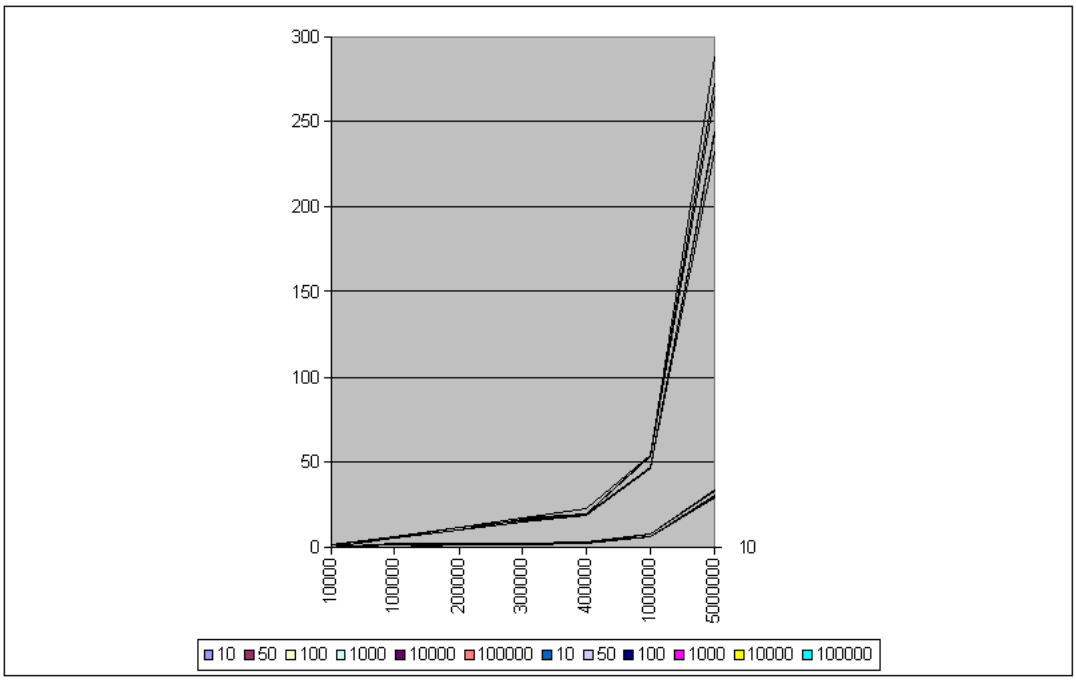


Figure 6.3: Performance

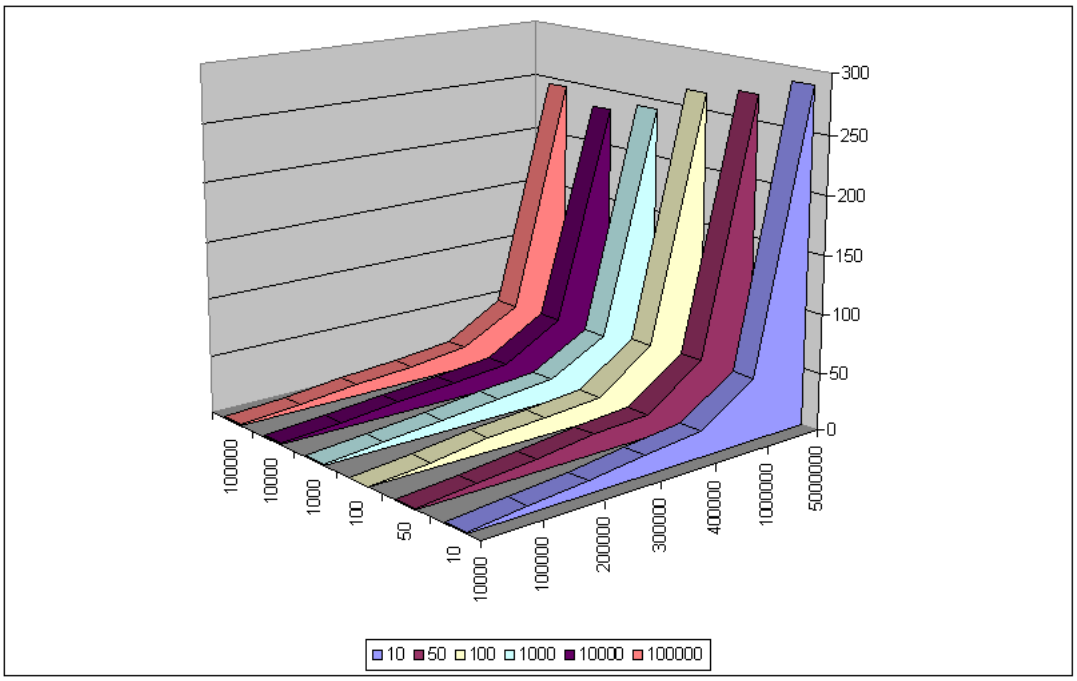


Figure 6.4: Performance with framework

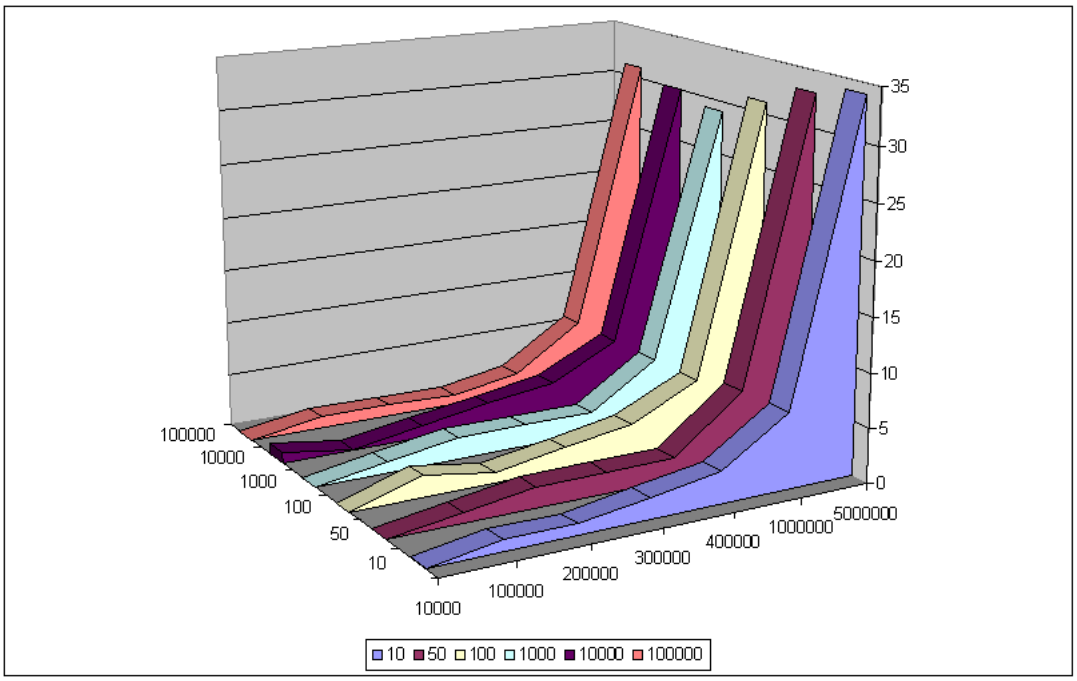


Figure 6.5: Performance with conventional synchronization

## 6.3 Future Possibilities

### 6.3.1 Deadlock detection

If further reduction of efficiency is not an issue, it is possible to implement a dynamic deadlock detection system. In Requirement 7 it was explained that there is a possibility to misuse the framework so that requests form a chain in which there will be a loop of requests that are pending for each other. See Figure 6.6 for an image of the proposal.

It would be possible for the framework to iterate through the whole chain of threads and requests via the callback pointers and verify that the last request does not point to any of the previous threads. In the figure, thread A sends a new request to thread B, which sends a new request to thread C. Next the deadlock is about to happen as request C attempts to send a request to thread A, see "1" in the figure.

At this point the framework should check the whole chain from A-C-B-A to find out that there is a conflict, and the program should assert. Alternatively, the framework could return an error value from sending a request as it cannot be served.

However, there is another, more complicated, scenario. On the right in the figure, thread A is serving a request which has added new request to thread B. Thread B in turn has added a new request to thread C. Next thread A sends a new request to thread D, which sends a new request to thread E. Next the deadlock is about to happen as request C attempts to send a request to thread E, see "2" in the figure. It

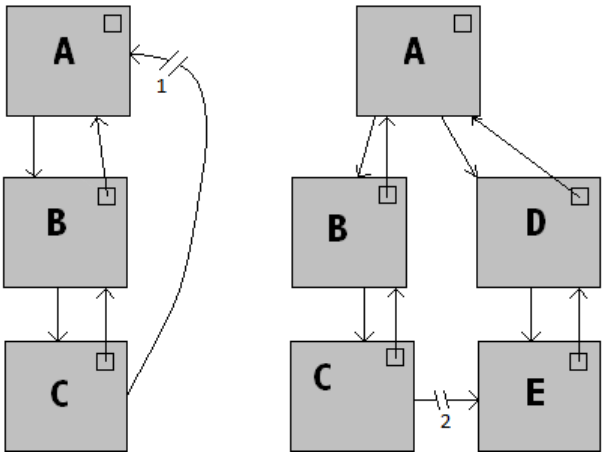


Figure 6.6: Dynamic deadlock detection

is possible to detect the conflict, but it requires more processing, thus causing more inefficiency.

If thread E is serving a request for D, the chain of callbacks is E-D-A. When compared to thread C's chain of callbacks, C-B-A, it can be seen that there will be a deadlock due to conflicting request A. The same algorithm applies also if thread C sent a request to E just before thread D's request.

If efficiency is critical, this kind of improvement should be added only as a debug feature so that constant checking were not performed in any production code. If it was added into production code, the code should not assert, but instead return an error when attempting to send a request. Still, this is not the kind of feature that should be used in production code as it is a sign of a programming error, not the kind of situation the application should attempt to recover from.

### 6.3.2 User-Kernel context switch

In the future, this framework could be converted to be used in an user-kernel context switch. However, the current status of the framework is lacking an important aspect which relates to security and robustness. The data from the user-thread to the kernel-thread should always be copied from user space to the kernel space. This can be done only by operating system calls as the processes cannot access each others' memory space directly.

Implementing the context switch might require a new limitation to the framework so that only copy-by-value objects could be passed over to the other process. If a copy-by-value approach would be taken, then all pointers could be converted to handles instead. The kernel process could then convert all the handles back to real memory locations. Naturally, this would only allow pointers into kernel-space.

There is a robustness issue with context switching as the kernel processes are considered to be vital to the operation of the system. If a kernel process faults then the whole system should shut down immediately in order to prevent any data corruption. This kind of robustness requirement should be a very heavy consideration when judging the framework's maturity and whether it's fit for such use.

A security aspect to consider is that if the kernel process is accessing a pointer which points to the user space directly, the user could, in its own thread, alter or de-allocate the memory that kernel process is accessing. Usually de-allocation causes the process to fault and execution is stopped. If the user thread simply alters the data that kernel is accessing, the user program can potentially take control of the operating system or cause arbitrary code to be run. Many vulnerabilities in modern operating systems are related to this kind of errors. All data used by the kernel should be copied to kernel-space with secure operating system calls.

User-kernel switch is not trivial, the framework should also take into account the payload within the requests that are being sent. If the data being passed is actually a pointer to data, then it is potentially impossible to copy the data behind a pointer to the kernel side. Besides, if a pointer is pointing to an array, which may be a very large array, the context switches become extremely heavy operations.

# 7. CONCLUSIONS

The thesis work was to implement a framework providing a versatile, easy to understand design that helps application developers use threads in C++ without using any synchronization or protection primitives.

The framework's APIs provide convenient abstraction from the thread handling APIs and, most importantly, removes the need for explicit use of mutexes and semaphores. It is possible to create complex multithreaded applications without introducing any mutexes, which can be considered as an improvement towards decreasing application complexity.

When implementing large applications, the framework makes source code easier to read, write and understand, as it forces a clear structure of requests and handlers. With smaller applications there is a somewhat large overhead from understanding the APIs and implementing many handlers. There are also some difficulties as the user has to define lots of different kinds of requests and request handlers that have almost identical naming. It is easy to create so many similarly named requests and handlers that it tends to become confusing.

It turns out that the implemented framework delivers practically all of the required features. The defined API does work as a generic solution for multithreaded communication, but it is not quite as simple as initially expected. Most of the complexity was caused by the usage of auto_ptr type in passing the data to the framework. Usage of auto_ptr requires constant type-casting and makes the code a bit harder to read.

All in all, the framework's performance with large data throughput is limited, but it delivers the care-free solution to protecting inter-thread data and provides an easy to extend interface for multithreading. The primary objective of the thesis work was to make multithreading easier for developers and the framework succeeds in that.

# BIBLIOGRAPHY

[1] Alexandrescu, A., Boehm, H., Henney, K., Hutchings, B., Lea, Doug., Pugh, B., Memory model for multithreaded C++: Issues. Open standards: 2006. URL *http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1777.pdf*. URL verified in April 2011.

[2] Barney, B., POSIX Threads programming, 2010. URL *https://computing.llnl.gov/tutorials/pthreads*. URL verified in April 2011.

[3] Boehm, H-J., Threads cannot be implemented as a library. Conference on Programming Language Design and Implementation: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation: Chicago: 2005. 7p.

[4] Chelf, B., Ensuring Code Quality in Multi-threaded Applications. How to Eliminate Concurrency Defects with Static Analysis. URL *http://www.coverity.com/*. URL verified in April 2011.

[5] Demerjian, C., Finally, a solution for the 64 core 4TB RAM market. The Inquirer, 2009. URL *http://www.theinquirer.net/inquirer/news/1137483/finally-solution-core-4tb-ram-market*. URL verified in April 2011.

[6] Gamma, E., Helm, R., Johnson, R., Vlissided, J., Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[7] Haikala, I., Järvinen H.-M. käyttöjärjestelmät, 2003. Talentum Media Oy.

[8] Java 2 Platform SE v1.4.2. Oracle, 2003, 2010. URL *http://download.oracle.com/javase/1.4.2/docs/api/java/lang/Thread.html*. URL verified in April 2011.

[9] Johnson, R., PTHREADS-WIN32. Open Source POSIX Threads for Win32. URL *ftp://sourceware.org/pub/pthreads-win32/dll-latest/README*. URL verified in April 2011.

[10] Koskimies, K., Mikkonen, T. Ohjelmistoarkkitehtuurit, 2005. Talentum Media Oy.

[11] Lo, J. and Eggers, S., Converting Thread-Level Parallelism to Instruction- Level Parallelism via Simultaneous Multithreading, ACM Transactions on Computer Systems, August 1997.

[12] Maraia, Vincent. The Build Master. First Edition, 2005. Addison-Wesley, 249 pages.

[13] Poiksalo, P-K., Digitaalitekniikan perusteet – aloitusopas digitaalisen maailman rakentajille, Dark Oy, 2003.

[14] Stability of the C++ ABI: Evolution of a Programming Language, Oracle Corporation and/or its affiliates, 2010. URL *http://developers.sun.com/solaris/articles/CC_abi/CC_abi_content.html*

[15] Stroustrup, B., The C++ Programming Language 3rd edition. Addison-Wesley, 1997.

[16] Symbian Developer Library. Nokia Corporation, 2011. URL *http://www.forum.nokia.com/Develop.* URL verified in April 2011.

[17] Symbian OS: Threads Programming. Nokia Corporation, 2005. URL *http://sw.nokia.com/id/8e18e213-3866-4567-bf1b-a04f938bb47e/Symbian_OS_Threads_Programming_v1_0_en.pdf.* URL verified in April 2011.

# APPENDIX 1: BASEREQUEST HEADER

```
/*
 * BaseRequest.h
 *
 * Created on: 11.4.2009
 * Author: teromaaranen */

#ifndef BASEREQUEST_H_
#define BASEREQUEST_H_
#include <vector>
using std::vector;
#include <deque>
using std::deque;
#include <iostream>

class BaseRequest {
public:
  /// aCallbackThread shall be NULL if no callback
  /// is expected
  BaseRequest(ThreadQueue* aCallbackThread = NULL)
    : iClientThread( aCallbackThread ),
    iCallbackRequired( aCallbackThread != NULL ),
    iCbArg( aCallbackThread
            ? aCallbackThread->iCurrentRequest
            : NULL ),
    iSubRequests() { }

  /// destructor for the request,
  /// no need to clean up anything
  virtual ~BaseRequest() { }

  /// the framework will call this function which will
  /// end up into the user component, DO NOT CALL THIS!
  virtual void act() = 0;
public:
  /// these members are public for convenience,
  /// DO NOT TOUCH THESE MEMBERS!
  /// sender of this request
```

```
  ThreadQueue* iClientThread;
  /// after the request is complete a callback
  /// shall be called
  bool iCallbackRequired;
  /// the item to which the callback is
  /// to be issued to, if needed
  ListItem<BaseRequest>* iCbArg;
  /// all the subrequests this request may have.
  /// Must be finished before this
  /// request is finished.
  vector<BaseRequest> iSubRequests;
};
#endif /* BASEREQUEST_H_ */
```

# APPENDIX 2: THREADQUEUE HEADER

```
/*
 * threadqueue.h
 *
 *  Created on: 11.4.2009
 *  Author: teromaaranen  */

#ifndef THREADQUEUE_H_
#define THREADQUEUE_H_
#ifdef WIN32
  #define WINDOWS
#endif
#ifdef WINDOWS
  #include "windows\pthread.h"
#endif
#include <semaphore.h>
#include <memory>
using std::auto_ptr;
#include <vector>
#include <string>
using std::string;
#include <iostream>
#include "listitem.h"

class ThreadQueue {
public:
  /// constructor
  ThreadQueue(string aName);
  /// virtual destructor
  virtual ~ThreadQueue();
  // user application needs to call Start to start
  // the new thread execution.
  void Run();
  /// kills the thread
  void Kill();
  // adds a request to the thread
  void Add(auto_ptr<BaseRequest>& aRequest);
protected:
```

```
  /// adds a subrequest to the currently ongoing request
  void AddSubRequest(auto_ptr<BaseRequest>& aRequest);
  /// returns the sent request for reading the callback
  auto_ptr<BaseRequest> ReadCallback()
    {
    return iCurrentRequest->ReadCallback();
    }
  /// starts the thread (main-loop)
  static void* Start(void* arg);
  /// ends the execution of the thread. This will
  /// return when the threadQueue is ready to be deleted.
  virtual void KillStart();
private:
  /// used for initialization
  void CreateMutexes();
  /// used for destroying the thread
  void DestroyMutexes();
  /// adds a callback to be expected
  void ExpectCallback();
  /// for debug purposes only
  inline virtual void PrintString(string aString)
  {
  /*std::cout << aString << std::endl;*/
  }
public:
  /// pointer to the request that is being currently served
  ListItem<BaseRequest>* iCurrentRequest;
private:
  /// name for this thread (for debug purposes only)
  string iName;
  /// all the requests that have been added to this thread
  ListItem<BaseRequest>* iList;
  ListItem<BaseRequest>* iListLast;
  /// false when the thread is not serving any requests
  bool iThreadIsIdle;
  /// should the thread be alive
  bool iAlive;
  /// the thread in which this component will run in
  pthread_t iThread;
```

```
  /// mutex for handling the request queue in the thread
  pthread_mutex_t iMutex;
  /// mutex for sleeping/waking up the thread
  pthread_mutex_t iThreadIsIdleMutex;
  };
#endif /* THREADQUEUE_H_ */
```

# APPENDIX 3: OS SPECIFICS HEADER

```
/*
 * os.h
 *
 *  Created on: 16.6.2010
 *  Author: teromaaranen
 */

#ifndef THREADQUEUE_OS_H
  #ifdef WIN32
    #ifndef WINDOWS
      #define WINDOWS
    #endif
  #endif
#endif
```